AFRL-RI-RS-TR-2011-282

SYSTEMS EXECUTION MODELING TECHNOLOGIES FOR LARGE-SCALE NET-CENTRIC DEPARTMENT OF DEFENSE SYSTEMS

Vanderbilt University

December 2011

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ AIR FORCE MATERIEL COMMAND ■UNITED STATES AIR FORCE ■ ROME, NY 13441

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2011-282 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/ /s/

WILLIAM MCKEEVER
Work Unit Manager

PAUL ANTONIK, Technical Advisor
Advanced Computing Division
Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved* **OMB No. 0704-0188**

1. REPORT DATE *(DD-MM-YYYY)* December 2011	**2. REPORT TYPE** Final Technical Report	**3. DATES COVERED** *(From - To)* December 2007 – May 2011

4. TITLE AND SUBTITLE SYSTEMS EXECUTION MODELING TECHNOLOGIES FOR LARGE-SCALE NET-CENTRIC DEPARTMENT OF DEFENSE SYSTEMS	**5a. CONTRACT NUMBER** N/A
	5b. GRANT NUMBER FA8750-08-1-0025
	5c. PROGRAM ELEMENT NUMBER 62702F
6. AUTHOR(S) Aniruddha Gokhale Douglas Schmidt Brian Dougherty Jules White	**5d. PROJECT NUMBER** 459T
	5e. TASK NUMBER VS
	5f. WORK UNIT NUMBER EM
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Vanderbilt University 110 21ST Avenue South Nashville TN 37203-2416	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RITA 525 Brooks Road Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI/RRS
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-RI-RS-TR-2011-282

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. PA# 88ABW-2011-6342
Date Cleared: 7 December 2011

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The "System Execution Modeling Technologies for Large-scale Net-centric DoD Systems (GUTS)" project developed and validated system execution modeling technologies that support design space exploration and evaluation of US Department of Defense (DoD) software-intensive net-centric systems before final implementation/testing and throughout subsequent system evolution. In particular, there were two primary objectives of the proposed project. First, the development of system execution modeling tools that enable system and software engineers to automatically and accurately detect, diagnose, and resolve system performance and stability problems stemming from design decisions made in early lifecycle phases. Second, apply these tools to systematically and empirically measure and evaluate the impact of multi-core and distributed processing, on software predictability and optimization in the context of representative DoD net-centric systems.

15. SUBJECT TERMS
Heuristic optimization; System Execution Modeling, Model based Software development

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** WILLIAM MCKEEVER
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	112	**19b. TELEPHONE NUMBER (*Include area code*)** N/A

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

Table of Contents

List of Figures

List of Tables

1. Summary

The "***System Execution Modeling Technologies for Large-scale Net-centric DoD Systems***" project developed and validated *system execution modeling* technologies that support design space exploration and evaluation of US Department of Defense (DoD) software-intensive net-centric systems *before* final implementation/testing and throughout subsequent system evolution. In particular, there were two primary objectives of this project. First, the development of system execution modeling tools that enable system and software engineers to automatically and accurately detect, diagnose, and resolve system performance and stability problems stemming from design decisions made in early lifecycle phases. Second, apply these tools to systematically and empirically measure and evaluate the impact of multi-core and distributed processing, as well as commercial off the shelf (COTS) componentization and Service-Oriented Architectures (SOAs), on software predictability and optimization in the context of representative DoD net-centric systems

The three year project, "System Execution Modeling Technologies for Large-scale Net-centric DoD Systems", conducted research to address these challenges, and resulted in practical artifacts as part of a tool suite called "GUTS: GEMS (Generic Eclipse Modeling System) Utilization Test Suite." The research conducted sought solutions along three dimensions:

- Design-time solutions – where research was conducted in model-based algorithms and technologies;

- Deployment-time solutions – where research was conducted on a variety of deployment-related optimization solutions, often based on heuristics;

- Run-time solutions – where research was conducted on runtime architectures to support the design-time and deployment-time decisions.

Over the course of the project, a significant effort was focused on deployment and configuration (D&C) challenges for next-generation of DoD-centric systems that leverage advances in computing, hardware and networking technologies. The project also applied the solutions to D&C challenges in cloud computing, including the use of multi-core technologies and mobile platforms.

This report describes the contributions accomplished during the three year project and is organized as follows: Section 2 provides an overview of system execution modeling; Section 3 describes the overall architecture of GUTS; Section 4 describes how model-driven engineering can help automate configurations of DoD-centric systems; Section 5 extends the work to demonstrate the idea for product line architectures; Section 6 focuses on an important topic for resource-constrained DoD-centric systems for which it is important to minimize the number of resources used to deploy DoD systems; Section 8 describes our effort on the deployment and configuration of distributed real-time and embedded (DRE) systems focusing on multi-objective deployment optimizations, i.e., when considering more than one parameter as a constraint in contrast to only one as described in Sections 6 and 7. Section 9 describes how model-driven technologies can be used to enable auto-scaling in cloud platforms and demonstrates how the model-driven technology developed under this effort, SCORCH, is used in power management for cloud environments; Section 10 describes our effort which continues the theme of auto-scaling issues in cloud computing, however, with a focus on investigations on battery power concerns for smart phone-based cloud computing environments; Section 11 furthers the theme on the auto-scaling issues in cloud computing with a focus on investigations into balancing the costs of acquiring and releasing a machine, and their reconfigurations. An additional dimension of this section is on the deployment and configuration of distributed systems, which can also handle deployment in the Cloud; Section12 focuses on the runtime infrastructure that enables the deployment and validation of DoD system artifacts in the runtime execution environment. Predictable performance of these capabilities is important for assuring QoS properties of DoD systems; lastly, Section 13 gives a short summary and discusses future work.

2. System Execution Modeling: Motivation and Overview

Our work in the Air Force's Global Information Grid (GIG), Army's Future Combat Systems (FCS) program, and the Navy's DDG 1000 program activities motivate a challenging problem facing developers of large-scale and layered DoD software-intensive net-centric systems: *how to discover, measure, and rectify structural, integration, and/or performance problems early in the system's lifecycle (e.g., in the architecture and design phases), as opposed to the final system integration phase, when mistakes are much harder and more costly to fix***.** The bulk of today's software technologies and validation/verification techniques are designed to develop and analyze relatively small-scale systems, where the set of tasks that will run in the system and their requirements for system resources are known in advance. Unfortunately, these technologies are poorly suited for large-scale DoD systems, where it is not possible to know the entire set of application tasks that will run on the system, the loads they will impose on system resources in response to a dynamically changing environment, or the order in which the tasks will execute. Moreover, even in today's smaller-scale DoD systems where load is known in advance, there is often little confidence that system quality of service (QoS) requirements will be met in the deployment phase due to the complexities of analyzing complex systems built atop commercial-off-the-shelf (COTS) hardware and software components.

Net-centric DoD systems must provide QoS support to process the right data, in the right place, at the right time, over a networked grid of computers. Some QoS properties required by these DoD systems include the low latency and jitter as expected in conventional real-time and embedded systems, and high throughput, scalability, and reliability as expected in conventional enterprise distributed systems. Achieving this combination of QoS capabilities in DoD systems is hard, particularly when the systems are developed using COTS hardware/software components.

Net-centric DoD systems are developed using applications composed of hardware/software components running on a wide range of platforms, some of which are COTS and some of which are proprietary legacy systems. These components are designed to provide reusable services to a range of application domains that are composed into domain-specific assemblies for application (re)use. The transition to modern methodologies, languages, and runtime platforms is intended to address problems of inflexibility and reinvention of core capabilities associated with prior monolithic, functionally-designed, and "stove-piped" legacy applications developed with the precise capabilities required for a specific set of requirements and operating conditions. Modern runtime platforms, such as component-based and Service-Oriented Architecture (SOA)-based systems, for example, are designed to have a range of capabilities that enable their reuse in other contexts. Moreover, these systems are developed in layers, e.g., layer(s) of infrastructure middleware services (such as naming and discovery, event and notification, security and fault tolerance) and layer(s) of application components that use these services in different compositions.

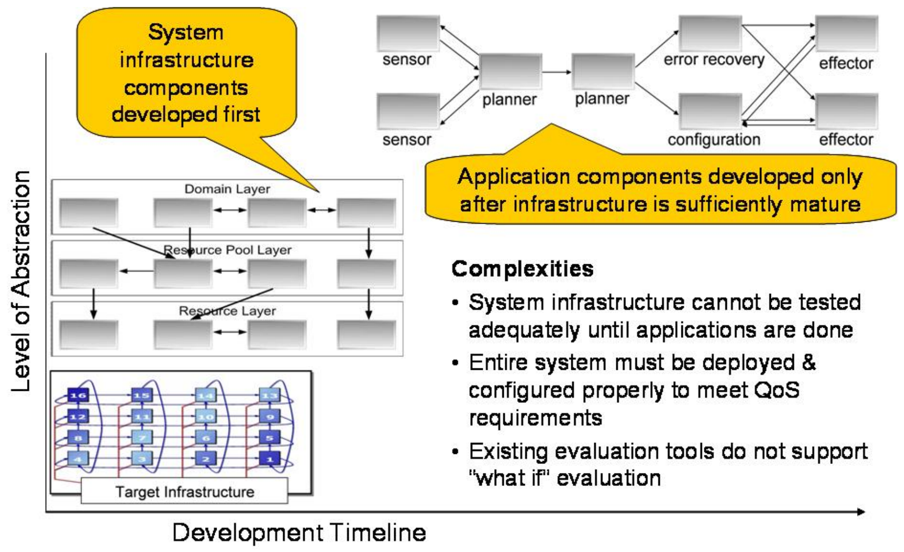

System infrastructure components developed first
sensor
sensor
planner
planner
error recovery
configuration
effector
effector
Application components developed only after infrastructure is sufficiently mature
Domain Layer
Resource Pool Layer
Resource Layer
Target Infrastructure
Level of Abstraction
Development Timeline
Complexities
• System infrastructure cannot be tested adequately until applications are done
• Entire system must be deployed & configured properly to meet QoS requirements
• Existing evaluation tools do not support "what if" evaluation

Figure 1: Complexities of Serialized Phasing in Net-centric DoD Systems

Figure 1 shows a particularly vexing problem facing researchers and developers of large and layered net-centric DoD systems, where the inadequacies of system architectures may not be ascertained until years into development. At the heart of this problem is the *serialized phasing* of layered system development, which postpones the discovery of design flaws that affect system QoS until late in the lifecycle, i.e., at final system integration time. A hallmark of serialized phasing is that application components are not created until *after* the completion of their underlying system infrastructure components, such as naming and discovery, event and notification, security and fault tolerance, and resource management.

As shown in Figure 1, in net-centric DoD systems built using serialized phasing, the implementations, configurations, and deployments of infrastructure components are often not tested adequately under realistic workloads until the applications are done. Moreover, both application and infrastructure components are hosted on the same target architecture, and therefore must be properly deployed and configured to achieve the desire QoS. As a result, it hard to know how well the system will satisfy key QoS properties due disconnects in the development of infrastructure and application components. Moreover, handcrafted software designs used in many DoD systems to address these concerns make it hard to conduct “what if” experiments on alternative system architectures and implementations to determine valid configurations to obtain performance goals for a particular workload. Making any significant changes to these types of handcrafted systems late in their lifecycle can be costly due to the impact on the design, implementation, deployment, and (re)validation of many application and infrastructure software/hardware components.

As a result of the serialized phasing problems described above, many performance characteristics of net-centric DoD system's components, such as invocation rates, time per invocation, and failover delay, are not known precisely in early phases of the lifecycle. The understanding of the component's timing and other properties is derived from the use of reusable software, such as COTS components and web services, as well as the influence of hardware based constraints (e.g. periodic sensor data, etc.). Although the individual component performance properties are often well understood, system architectures and designs routinely fail to meet their end-to-end QoS goals due to unforeseen complexities of system integration. The major forces that make predicting end-to-end QoS hard in net-centric DoD systems include the following:

- **Emergent complexity.** Although the individual QoS properties of components may be documented, unforeseen side effects of their interaction, such as prolonged execution time because of competition for CPU usage with other collocated components or event queuing in lower priority components due to high event rates in higher priority components, are not realized until integration time. It is therefore hard to accurately predict end-to-end QoS until the system is integrated and tested in its target environment.
- **Ripple effects.** Individual components yield different QoS metrics for different hardware/software configurations. For example, a CPU bound component will perform better (e.g., produce a lower service time) on a faster CPU, whereas a network bound component will perform better with a larger bandwidth. Moreover, different versions of software may introduce new optimizations (e.g., improved caching or remote invocations) that can alter existing QoS metrics and affect end-to-end QoS. It is hard to predict precisely how end-to-end QoS will be affected by such changes until the system is integrated and tested in its target environment.
- **Vast configuration spaces.** System components can have multiple configurations (e.g., setting of attributes) that can affect its end-to-end QoS. For example, a component that permits configuring the number to threads handling input event will produce different performance metrics based on the number of threads configured. Manually trying to understand how different configurations of system components affect end-to-end QoS is time consuming, tedious, and error-prone.
- **Hardware interaction.** End-to-end system performance is highly dependent on the deployment topology of the application. Different deployment topologies can have widely varying end-to-end execution times. Understanding how colocation/distribution decisions, the interaction of different COTS/proprietary hardware and software components, and network topologies will affect timing properties is infeasible for systems of realistic sizes containing hundreds or more components.

As a result of the forces described above, manually deriving the most promising deployment configurations for net-centric DoD systems is infeasible [1]. These types of systems often have numerous types of constraints and huge solution spaces that make it hard to manually derive valid deployments. Moreover, net-centric DoD system constraints, such as network bandwidth and memory footprint limitations, are exponential in complexity and require constraint solvers (e.g., finite domain solvers) [2], [3] and numerical methods (e.g., the simplex method) [4] to find valid deployments.

To address the challenges of serialized phasing in DoD systems described above, DoD researchers and practitioners need a methodology and an associated suite of system execution modeling (SEM) tools [5] that use model-driven engineering (MDE) technologies [6], [7] to simplify the following:

- **Emulation of application component behavior** in terms of computational workloads, resource utilizations and requirements, and network communication. This step should be done quickly and precisely, e.g., by using domain-specific modeling languages (DSMLs) [8–10] that capture the behavior and workload of system components at a high-level of abstraction and then generate code that executes emulated components in a representative execution environment (ideally the actual runtime deployment platform).
- **Configuration, deployment, and execution of the emulated application components** atop actual infrastructure components to determine their impact on QoS empirically in actual run-time environments. These steps should also be accomplished quickly and precisely, e.g., by using DSMLs to specify realistic deployment configurations and synthesize metadata describing them. This metadata can then be processed by the same deployment and configuration tools as the actual system, with SEM tools providing mechanisms to record, consolidate, and collect QoS metrics (such as execution times and resource usage) from the runtime environment.
- **Process of feeding back the results to enhance system architectures and components** to improve end-to-end QoS. This step should be accomplished by archiving the collected QoS metrics and providing tools that view and analyze the overall results of a deployment. SEM tools should also provide histories of the collected metrics to enable engineers and architects to understand performance and make well-informed decisions to improve QoS.

Figure 2 shows the relationships between the steps described above. Over time as the actual application components mature, they can replace the emulated components providing a more realistic evaluation environment for DoD systems, thereby alleviating many of the current problems with serialized phasing in large-scale software-intensive systems.

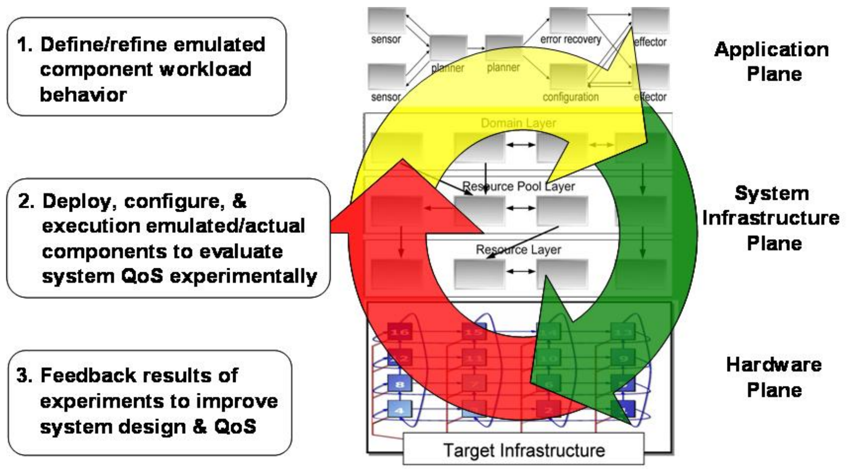

1. Define/refine emulated component workload behavior
2. Deploy, configure, & execution emulated/actual components to evaluate system QoS experimentally
3. Feedback results of experiments to improve system design & QoS
sensor
planner
error recovery
effector
configuration
Domain Layer
Resource Pool Layer
Resource Layer
Target Infrastructure
Application Plane
System Infrastructure Plane
Hardware Plane

Figure 2: Evaluating the QoS of DoD Systems via System Execution Modeling Tools

In general, SEM tools enable system engineers, software architects/developers, and quality assurance (QA) engineers to address the inherent complexities that arise from properties of production systems, including communication delay, temporal phasing, parallel execution, and synchronization. There are typically only a few deployment configurations that actually can satisfy the functional and performance requirements established in software and system architecture. SEM tools enable architects and engineers to discover, measure, and rectify incipient integration and performance problems early in a system's lifecycle (e.g., in the analysis and/or design phases). These tools help shift the focus of the software integration resources to productive activities that evaluate and validate system performance and end-user value, rather than serving as the de facto system design debugging activity, as is often the case today.

3. Architecture of GUTS

To achieve these objectives of SEM and to address the challenges described in Section 2, we have prototyped and empirically validated the GEMS (Generic Eclipse Modeling System) Utilization Test Suite (GUTS), which is a SEM tool suite consisting of performance modeling languages [11], workload emulators [12], and constraint solvers [13] that can automatically derive valid deployment configurations (such as mappings to various multi-core, multi-processor, and multi-node configurations), generate end-to-end system performance tests, and autonomously/intelligently explore the deployment solution space of net-centric DoD system designs.

3.1 GUTS Workflow

GUTS enables developers and testers to annotate system architectures with the expected workload of components and then to empirically test the end-to-end performance of the design in multiple deployment configurations. The key capabilities of GUTS that allow system engineers, software architectures, and software developers to design and test net-centric DoD systems include:

- **Capability 1**: **Synthesis of faux components** that have the same interfaces and attributes as the real components, but there is implementation derived from the constructed performance models. The faux components are then instrumented in the target environment to observe and collect QoS metrics, such as throughput, service time, or end-to-end deadline. This capability will allow developers and testers to conduct integration tests during early stages of development using emulated components to locate and rectify design flaws that may be too costly to fix later in the development lifecycle.
- **Capability 2. Continuous system integration** is achieved by allowing real components to replace the faux components as they are completed. This capability will allow developers to continuously test their system in the target environment and produce more realistic QoS metrics. Moreover, it will allow developers to test their components under realistic and hypothetical workloads to better understand how the system's QoS is affected.
- **Capability 3. Visual analysis of QoS metrics** collected while the system is executed in its target environment. This capability will help to alleviate the time and effort of presenting the collected metrics in a clear and easy to understand format.
- **Capability 4. Constraint solvers that can automatically satisfy the complex deployment constraints**, such as resource constraints, and generate valid configurations. Moreover, the deployment solution space can be searched using heuristics allowing the tool to look for deployments considering high in quality. This capability allows the tool to systematically iterate through deployments and test a system design in the most promising deployment configurations.
- **Capability 5. An autonomous testing mode** in which the tool repeatedly solves for valid deployment plans, executes end-to-end performance tests on it, and then uses the performance results to refine the deployment topology. If a deployment topology is found to satisfy the end-to-end goals, the iterative performance testing stops. If not, the system

continues refining the deployment plan and running tests until it finds that it cannot improve upon its current solution.

By allowing developers to evaluate a system design empirically in multiple deployment configurations using both faux and real components, GUTS allows developers and testers to begin conducting integration tests during the early stages of development. Applying GUTS throughout the system lifecycle will help pinpoint and resolve design flaws earlier in the development lifecycle before they become too hard and costly to locate and rectify. More importantly, GUTS will assist system developers and testers in searching the deployment and configuration space for net-centric systems to locate valid developments and configurations that meet their desired QoS requirements.

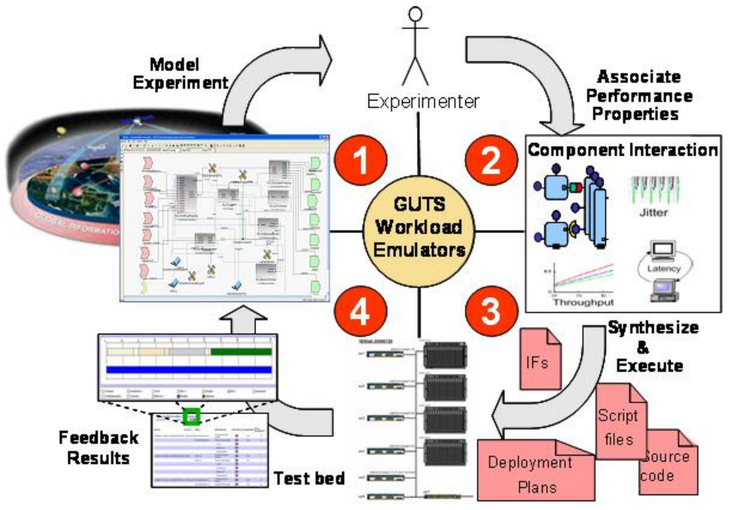


Figure 3: Workflow of Applying GUTS to Evaluate Net-centric DoD System Performance

Figure 3 shows an example workflow of applying GUTS to evaluate the performance of a representative DoD system, such as an information dissemination application running in the tactical GIG. The workflow in Figure 3 consists of the following steps:

1. Quality assurance (QA) engineers use the GUTS SEM tool to work with system engineers and software architects early in the system lifecycle to compose scenarios that specify the structure of the system (e.g., the component and their interconnections) and exercise critical system paths.

2. QoS engineers use GUTS to work with QA engineers to create representative scenarios that associate performance properties with (a) individual components, such as a component's CPU utilization or (b) the system as a whole, such as the deadline of a critical path through the system.

3. The information captured in the first two steps is then synthesized into executable code and configuration metadata, then QA engineers use GUTS to configure the workload emulators to run experiments, generate deployment plans, and measure performance along critical paths on the target architecture.

4. QA engineers work together with system engineers and software architects to explore design alternatives from multiple computational and valuation perspectives to (1) analyze results to verify if deployment plan and configurations meet performance requirements and (2) quantify the costs of certain design choices on end-to-end system performance. The results can be fed back to the software architects and developers to address any problems that were identified using GUTS.

In the context of net-centric DoD systems, our proposed GUTS SEM tool will help systems engineers and software architects conduct "what if" experiments to discover, measure, and rectify performance problems early in the lifecycle (e.g., in the architecture and design phases), as opposed to waiting until the integration phase, when mistakes are much harder and more costly to fix.

3.2 **GUTS Architecture**

As shown in Steps 1-4 in the workflow in Figure 3, various roles coordinate on the design and evaluation of a net-centric system and produce multiple domain-specific models. For example, in Step 1, developers produce architectural models of the system and in Step 4 developers leverage another model to explore the solution space of the design. A key capability for coordinating the development of the set of models from the workflow in Figure 3 and achieving the autonomous testing capability described in Section 3.1 is a method for maintaining the consistency of multiple independent models.

For example, when new simulation results are produced from experiments, the results may invalidate the currently modeled assumptions about the system's resource consumption behavior. In this case, the autonomous testing facility must update both the behavioral specification of the system and the deployment models. Updating the deployment model will result in the constraint solver producing a modified deployment that must then be fed into the architectural model of the system to produce new deployment descriptors. As each model changes, multiple other models must be kept in sync.

To manage the complexity of leveraging multiple models of a system, we have developed a prototype of GUTS that uses the model event bus shown in Figure 4 to integrate multiple domain-specific modeling languages (DSMLs) that capture different concerns, such as system behavior, workload, and deployment and configuration.

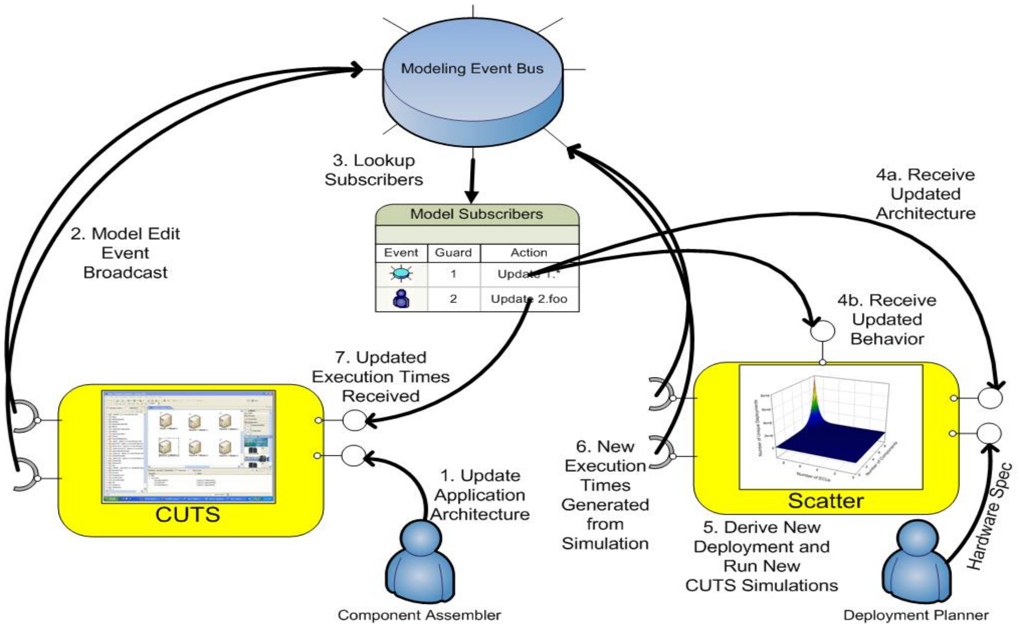

Modeling Event Bus
2. Model Edit Event Broadcast
3. Lookup Subscribers
Model Subscribers
Event
Guard
Action
1
Update 1.*
2
Update 2.foo
4a. Receive Updated Architecture
4b. Receive Updated Behavior
7. Updated Execution Times Received
CUTS
1. Update Application Architecture
Component Assembler
6. New Execution Times Generated from Simulation
Scatter
5. Derive New Deployment and Run New CUTS Simulations
Hardware Spec
Deployment Planner

Figure 4: The GUTS Model Event Bus

In Step 1 of Figure 4, developers specify the architecture of the system through structural models. Developers also produce a model of the expected behavior of the individual components in the system in Step 1. The behavioral specification is used to help synthesize faux components for experiments as outlined in Capability 1 of Section 3.1. The updates to the system behavior cause an event to be broadcast to the model event bus in Step 2. The model event bus uses an autonomous service to determine which models are potentially affected by the event and to notify the relevant models of the change.

The new behavioral data has two ramifications on the Scatter component. First, a new deployment plan must be deduced (Capability 4 in Section 3.1) that properly accounts for the updated information on each component's resource consumption. Second, new faux components must be synthesized to model the modified behavior (Capability 2 in Section 3.1). Once these updates to the deployment model and faux components are complete, new experiments must be run to understand the implications of the changes on the system as a whole. The output of these experiments is returned to the data (Capability 3 in Section 3.1). After every iteration developers may replace one or more faux components with actual implementations allowing the continuous integrating and testing of the system as a whole (Capability 2 in Section 3.1).

The output of the experiments may in turn show that certain key end-to-end deadlines are not being met and that the system's architecture must be modified. The modification of the system's architecture will in turn create another ripple of changes throughout the system that must be integrated into the models, emulated, and evaluated. The model event bus is a key enabler of the continuous autonomous integration and testing in GUTS (Capability 5 of Section 3.1).

As shown in Figure 4, each DSML can be viewed as a component in GUTS. DSMLs we used for the GUTS prototype include the following:

The *Component Workload Emulator (CoWorkEr) Utilization Test Suite* (CUTS) [12], which is a MDE tool that allows developers to rapidly create emulations of component-based applications and understand is performance in the target domain well before system integration The CUTS component encompasses three different DSMLs, including the *Platform Independent Component Modeling Language* (PICML) [14], [15], *Component Behavior Modeling Language* (CBML) [16], *Workload Modeling Language* (WML) [16]. PICML is used to capture the structural aspect of component-based applications, such as provided and required interfaces, event sources and sinks, and attributes. CBML is designed to capture the behavior of reactive systems, such as component-based systems. WML is used to associate workload parameters with the actions in CBML to create operations that can be emulated on the target architecture (e.g., a representative testbed used for system integration testing).

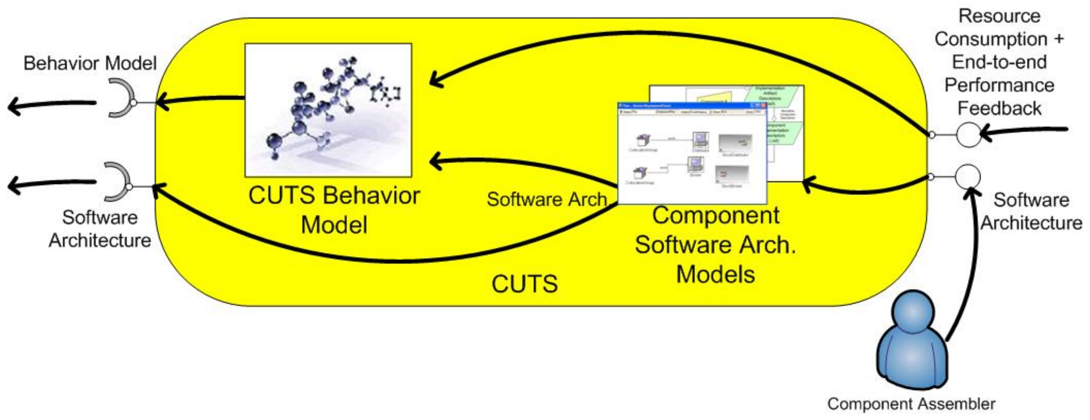


Figure 5: The CUTS DSML and Related Tools and DSMLs

Figure 5 shows how CUTS models can be used to specify the behavior and workload of individual components so that the system can be emulated on the target architecture (i.e., a representative testbed used for system integration) during the early stages of development (i.e., before system integration). As illustrated in Figure 5, CUTS models is composed of component software architecture models captured using PICML and behavior models for each component captured using CBML and WML. Initially, a developer bootstraps the CUTS models by defining the behavior and workload parameters (e.g., how much memory to consume) and estimating resource usage of each component in the system. These initial estimates are then replaced after the first emulation of the system and the CUTS component receives resource usage values as feedback via the CUTS runtime framework.

Scatter [13], which is an automated deployment planning tool based on constraint logic programming techniques. Scatter takes as input a model of a set of components, their functional requirements (e.g. OS type, required libraries, and collocated components), their resource consumption values, and a set of deployment targets (e.g. embedded microprocessors.) and produces a valid assignment of components to nodes (a deployment plan). Moreover, if multiple valid deployment plans exist, Scatter can attempt to find an optimized deployment plan with a user provided a cost function.

Figure 6 shows Scatter and its related tools, which include the Scatter solver and Scatter's interface to the CUTS component emulation environment. First, the deployment planner provides Scatter the list of available hardware resources for running the components specified in the software architecture. Second, Scatter invokes its constraint solver-based deployment planner to derive a valid deployment of components to nodes. Scatter then invokes the CUTS emulation environment to test the newly derived deployment plan and producing new performance data. The performance data is then fed back either directly to developers (to modify the system architecture) or to Scatter to attempt to improve the deployment plan.

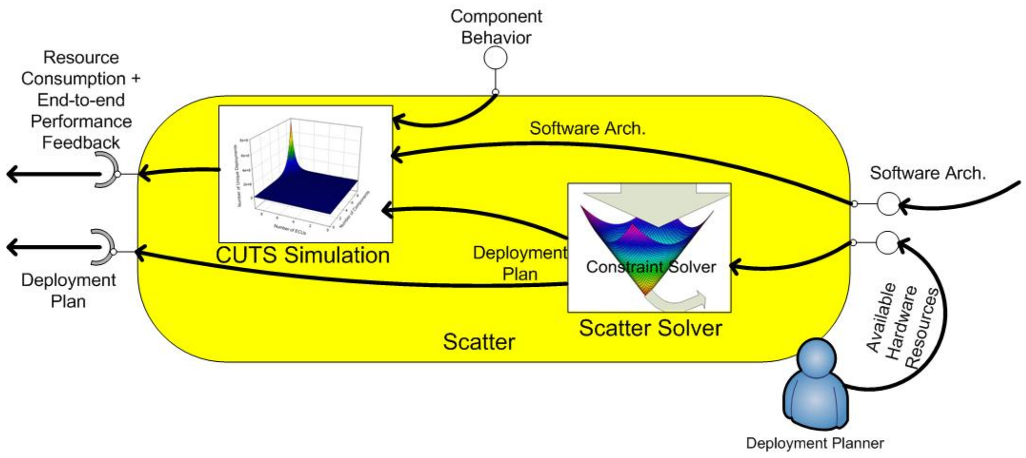


Figure 6: Scatter and Related Tools

The Scatter component receives the list of components as input, their modeled resource consumptions, and the available hardware resources. The Scatter solver uses this information to produce a new deployment plan designed to improve the end-to-end performance of the system's key critical paths. The new deployment plan produced by Scatter is used to create a new model of the system, such as a PICML deployment and configuration based on new resource consumption estimates, in CUTS. CUTS then emulates the new model and produces updated end-to-end execution times and resource consumption values. The new emulation results are returned as feedback to the original CUTS behavior models to refine the understanding of the system's execution properties.

A GUTS-based model can receive (1) human-based input, such as a developer pointing and clicking to create modeling elements, and (2) machine-based input, such as the output of simulation results. For example, the CUTS behavior model can receive human-input in the form of a human manually modeling the behavior specification of individual components, and machine-input in the form of resource consumption values measured while emulating the system on the target architecture.

Viewing each model as a component with input and output ports allows the seamless integration of new DSMLs, such as a DSML for producing queuing models from emulation results measured by CUTS, into GUTS to enhance multi-model collaboration. As shown in Figure 4, the multiple components (i.e., DSMLs) used by GUTS are integrated via a model event bus. This event bus receives notifications when the output of a model changes, such as when new simulation results are generated. When a new event arrives, a lookup table is used to determine which subscribers are registered for the event. Guards can also be associated with a subscriber to filter the events sent.

3.3 GUTS in the Context of the AFRL SPRUCE Program

We have conducted several experiments on our GUTS SEM tools described in Section 3.2 using the Systems and Software PRoducibility Collaboration and Experimentation Environment (SPRUCE) testbed developed as part of the OSD sponsored, AFRL executed, Software and Systems Test Track (SSTT) program. The objective of SPRUCE is to provide a distributed, open collaborative R&D environment were software-intensive systems productivity initiative (SISPI) researchers can demonstrate, evaluate, and document the ability of their tools, methods, techniques, and run-time technologies to yield affordable and predictable production of software-intensive systems. SPRUCE provides the following four key capabilities to achieve this objective:

- Ability to define and collaborate around challenge problems distilled from representative DoD acquisition programs.
- Ability to define and collaborate around promising candidate solutions to said challenge problems.
- Ability to define, conduct and collaborate around realistic experiments related to challenge problems and candidate solutions.
- Ability to shepherd technology transition across the software producibility spectrum and across operational domains.

To provide these capabilities, SPRUCE features a collaboration environment for bringing program engineers and SISPI researchers together to develop, experiment, and transition technologies. Figure 7 shows how the GUTS SEM tools relate to the broader SPRUCE technology identification, creation, and transition process. The benefits of conducting experiments on our SEM tools within the SPRUCE environment include (1) evaluating the efficacy of GUTS in a representative “at-scale” environment, (2) serving as an early adopter exemplar to encourage other researchers to leverage SPRUCE to showcase their research activities, and (3) providing the opportunity to collaborate more effectively with other researchers, DoD system integrators, and COTS technology suppliers interested in the work.

We have conducted evaluations and experiments throughout the effort. Initial experiments conducted early in the project evaluated the candidate technology elements. The experiments were defined in collaboration with SPRUCE collaborators and challenge problems specified on the SPRUCE portal. Subsequently the experiments were executed to collect and evaluate (1) functional and QoS attributes provided by the candidate technologies compared to net-centric system requirements (such as the tactical), (2) relative performance of a candidate technology compared to other candidate technologies, (3) coverage of use cases by the candidate technology, and (4) richness of application programmatic interfaces for the candidate technology. The results of these experiments were used to evaluate individual technologies, select among candidate technologies, design integration strategies, and identify needed enhancements to the candidate technologies.

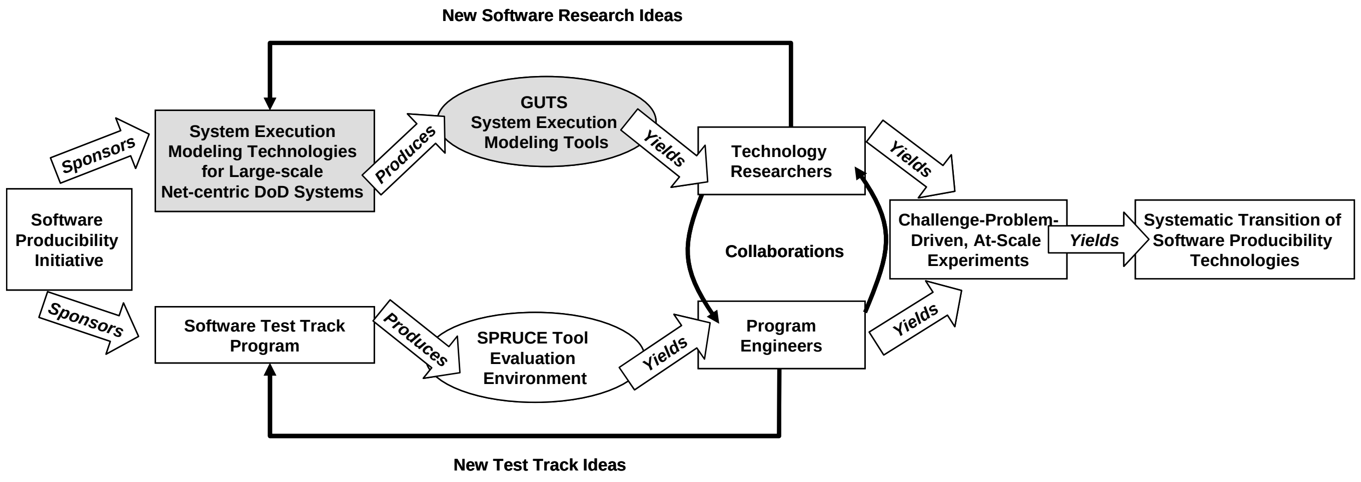

New Software Research Ideas
Sponsors
System Execution Modeling Technologies for Large-scale Net-centric DoD Systems
Produces
GUTS System Execution Modeling Tools
Yields
Technology Researchers
Yields
Software Producibility Initiative
Collaborations
Challenge-Problem-Driven, At-Scale Experiments
Yields
Systematic Transition of Software Producibility Technologies
Sponsors
Software Test Track Program
Produces
SPRUCE Tool Evaluation Environment
Yields
Program Engineers
Yields
New Test Track Ideas

Figure 7: GUTS in the Context of AFRL SPRUCE

We have set up the necessary experimentation infrastructure to support the experiments at one or more of several testbeds available to us under this project. Our testbed shown in Figure 8 is the ISISlab at Vanderbilt University (www.dre.vanderbilt.edu/ISISlab), which is a dedicated cluster of over 100 high-end x86 CPUs capable of running multiple operating systems, such as many versions of Linux, Windows, Solaris, Mac OSX, and BSD UNIX. We have recently enhanced this cluster with control software from Emulab, an NFS-sponsored testbed at the University of Utah. The right hand side of Figure 8 shows the hardware and networking elements in ISISlab, which are also part of the Spirals 0 and 1 Experiment Infrastructure for the SPRUCE project.

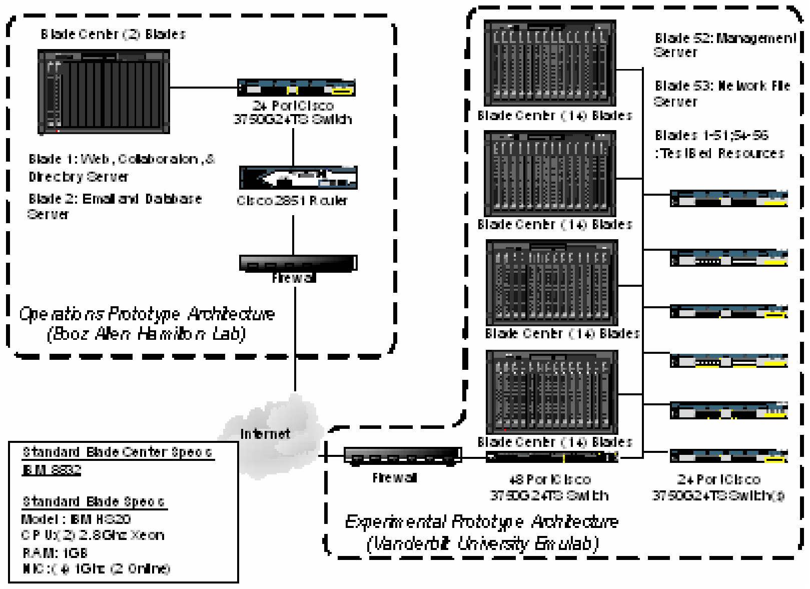

Blade Center (2) Blades
24 Port Cisco 3750G24TS Switch
Blade 1: Web, Collaboration, & Directory Server
Blade 2: Email and Database Server
Cisco 2851 Router
Firewall
Operations Prototype Architecture (Booz Allen Hamilton Lab)
Blade Center (14) Blades
Blade Center (14) Blades
Blade Center (14) Blades
Blade Center (14) Blades
Blade 52: Management Server
Blade 53: Network File Server
Blades 1-51;54-56 :TestBed Resources
Internet
Standard Blade Center Specs
IBM 8832
Standard Blade Specs
Model: IBM HS20
CPU:(2) 2.8Ghz Xeon
RAM: 1GB
NIC:(4) 1Ghz (2 Online)
Firewall
48 Port Cisco 3750G24TS Switch
24 Port Cisco 3750G24TS Switch(s)
Experimental Prototype Architecture (Vanderbilt University Emulab)

Figure 8: Testbed Environment for GUTS

4. Automating Configurations using Model-driven Engineering

This section describes the role of model driven engineering in automating configurations for DoD-centric systems.

4.1 Introduction

Distributed real-time embedded (DRE) systems (such as avionics systems, satellite imaging systems, smart cars, and intelligent transportation systems) are subject to stringent requirements and constraints. For example, timing constraints require that tasks be completed by real-time deadlines. Likewise, rigorous quality of service (QoS) demands (such as dependability and security) require a system to recover and remain active in the face of multiple failures [17]. In addition, domain-specific constraints (such as the need for power management in embedded systems) must be satisfied. To cope with these complex issues, applications for DRE systems have traditionally been built from scratch using specialized, project-specific software components that are tightly coupled with specialized hardware components [18].

To reduce development cycle-time and cost, the new generation of DRE systems are increasingly being developed by *configuring* applications from multiple layers of commercial-off-the-shelf (COTS) hardware, operating systems, middleware components, and mission level software components [19]. These types of DRE systems require the integration of 100's-1,000's of software components that provide distinct functionality (such as I/O, data manipulation, and data transfer) that must work in concert with other software to accomplish mission-critical tasks (such as self-stabilization, error notification, and power management). The software configuration of a DRE system directly impacts its performance, cost, and quality.

As COTS-based DRE systems increase in size and complexity the traditional design techniques based on complete in-house proprietary construction are not sufficient. Moreover, the traditional techniques cannot configure COTS-based DRE systems that can simultaneously address the stringent requirements and constraints outlined above [20]. The objective of DRE system configuration is to determine exactly what combination of hardware/software components will provide the requisite QoS. In addition, the combined purchase cost of the components cannot exceed a predefined amount, referred to as the project budget.

An overall DRE system configuration can be split into a software configuration and a hardware configuration. A valid software configuration must meet all real-time constraints (such as minimum latency and maximum throughput), provide required functionality, and also satisfy all domain-specific design constraints while not exceeding the available budget for purchasing software components. Similarly, the hardware configuration must meet all constraints without exceeding the available hardware component budget.

It is likely that for each portion of desired software functionality, there are multiple COTS components that can perform the desired function. Each option differs in QoS provided, the amounts/types of computational resources required, and purchase cost. The complexity associated with DRE systems composed of 100's-1,000's of components makes it hard to find configurations that meet complex QoS constraints. Creating and maintaining error-free configurations is also hard due to the large number of complex configuration rules and QoS requirements.

Solution approach → Model-driven automated configuration techniques. This section presents techniques and tools that leverage the *Model Driven Architecture* (MDA) paradigm to determine valid DRE system configurations that fit budget limits. To help address the difficulty of DRE system configuration, DRE system designers can use MDA to visualize configuration options/rules, verify configuration validity, and compare potential DRE system configurations.

MDA is a design approach for specifying system configuration constraints with platform-independent models (PIMs). Each PIM can be used as a meta-modeling for constructing platform-specific models (PSMs) [21]. These PSMs can be analyzed to determine DRE system configurations that meet budget constraints. After a PSM is determined for implementation, it can be used as a blue print for constructing an actual DRE system implementation that meets all design constraints specified within the PIM [7]. As DRE system requirements evolve and additional constraints are introduced, the PIM can be modified and new PSMs constructed. Systems that are constructed using these PSMs can reflect additional constraints and requirements more readily than those developed manually.

We conducted a survey of Meta-modeling techniques for creating DSMLs that can be applied to DRE system configuration and demonstrated the creation of a DSML for modeling hardware/software component options, resource constraints, and budgetary constraints. We also show how to utilize modeling environments to create models that adhere to the DSML for DRE system configuration. In addition, we demonstrate an interpreter that can examine models of hardware/software DRE system configuration options, generate code, which can ultimately be used to produce output models that provide valid, high-quality large-scale DRE system configurations.

4.2 Large-scale DRE System Configuration Challenges.

This section presents the criteria for valid DRE system configurations, describes the challenges that make determining configurations hard, and provides a survey of current techniques and methodologies for DRE system configuration. Software and hardware components often have complex interdependencies on the consumption and production of resources (such as processor utilization, memory usage, and power consumption). An overall DRE system configuration consists of a valid hardware configuration and valid software configuration in which the computational resource needs of the software configuration are provided by the computational resources produced by the hardware configuration. If the resource requirements of the software configuration exceed the resource production of the hardware configuration, a DRE system will not function correctly and is considered invalid.

Consider configuration options for a satellite imaging system case study we used. This DRE system consists of two software components: an image processing algorithm and software that defines image resolution capabilities. There are multiple options for each software component, each of which provides a different level of service. For example, there are three options for the image resolution component. The high-resolution option offers the highest level of service, but also requires dramatically more RAM and CPU to function than the medium or low-resolution options. If the resource amounts required by the high-resolution option are not supplied, then the component cannot function, preventing the system, as a whole, from functioning correctly. If RAM or CPU resources are scarce, the medium or low-resolution option should be chosen. However an additional design constraint may require at least medium image resolution. Assuming sufficient resources for only the medium and low-resolution options, the only option that satisfies all constraints is the medium image resolution option.

Further, the inclusion of a component in a configuration may prohibit or require the use of one or more components. Certain software components may have compatibility problems with other components. For example, each of the image resolution components may be a product of separate vendors. As a result, the high and medium-resolution component may be compatible with any image processing component while the low-resolution component may only be compatible with image processing components made by the same vendor. These compatibility issues add another level of difficulty to determining valid DRE system configurations.

Large-scale DRE systems may consist of many software and hardware components with multiple options for each component, resulting in an exponential number of potential configurations. Due to the multiple design, real-time, and resource constraints discussed earlier, it is simple to see that arbitrarily selecting components for a configuration is ineffective. That is, the huge magnitude of the solution space prohibits the use of manual techniques. Automated techniques, such as Constrained Linear Programming (CLP), use Constraint Satisfaction Problems (CSPs) to represent system configuration problems [22][23]. These techniques are capable of determining optimal solutions for small-scale system configurations but require the examination of all potential system configurations. Techniques utilizing CSPs are ideal, however, for system configuration problems involving a small number of components as they can determine an optimal configuration (should one exist) in a short amount of time.

The exhaustive nature of conventional CSP-based techniques, however, renders them ineffective for large-scale DRE system configuration. Without tools to aid in large-scale DRE system configuration, it is a struggle for designers to determine *any* valid large-scale system configuration. Even if a valid configuration is determined, other valid system configurations may exist with vastly superior performance and dramatically less financial cost. Further, the constant development of additional technologies, legacy technologies becoming unavailable, and domain specific and design objectives constantly in flux, valid configurations can quickly become invalid, requiring that new configurations be discovered rapidly. It is thus imperative that advanced design techniques are developed to identify and validate large-scale DRE system configurations.

The following is a summary of key challenges that make it particularly hard to configure DRE systems using COTS components:

- **Choosing between multiple levels of service.** Software components provide differing levels of service. For example, a designer may have to choose between three different software components that differ in speed and throughput. In some cases, a specific level of service may be required, prohibiting the use of certain components.

- **Complex resource interdependencies.** Hardware components provide the computational resources that software components require to function. If the hardware does not provide an adequate amount of each computational resource, some software components cannot function. An overabundance of resources indicates that some hardware components have been purchased unnecessarily, wasting funds that could have been spent to buy superior software components or set aside for future projects.

- **Satisfying differing resource requirements.** Each software component requires computational resources to function. These resource requirements differ between components. Often, components offering higher levels of service require larger amounts of resources and/or cost more to purchase. Designers must therefore consider the additional resulting resource requirements when determining if a component can be included in a system configuration.

- **Meeting budgetary constraints.** Each component has an associated purchase cost. The combined purchase cost of the components included in the configuration must not exceed the project budget. It is therefore possible for the inclusion of a component to invalidate the configuration if its additional purchase cost exceeds the project budget regardless of computational resources existing to support the component.

- **Choosing from many components.** Large-scale DRE systems can require hundreds to thousands of components to function. For each component there may be many options available for inclusion in the final system configuration. Due to the complex resource interdependencies, budgetary constraints, and domain- specific design constraints it is hard to determine if including a single component will invalidate the system configuration. Even automated techniques require years or more to examine all possible system configurations for such problems.

4.3 Reviewing Literature on System Configuration Optimization.

This section surveys various DRE system configuration problems and model analysis techniques, including hardware/software co-configuration problems and heuristic algorithms for resource-constrained configuration. We describe several different types of DRE system configuration problems. We also examine several techniques that have been applied to aid in the determination of optimal or near-optimal DRE system configurations, such as Constrained Linear Programming and Constraint Satisfaction Problems. Finally, we will describe how MDA can be used to mitigate many of the problems associated with these techniques.

Feature models are visual diagrams that have been used to model Software Product Lines (SPLs) as well as system configuration problems. Czarnecki et al. use feature models to describe the configuration options of systems [24]. Feature models are represented using tree structures with lines, representing functional constraints that are connected to the various candidates for inclusion in an SPL, known as features. The feature model uses functional constraints to illustrate the effects that selecting one or more features have on the validity of selecting other features. As a result, it is obvious if the inclusion of a feature will result in an invalid system configuration. Czarnecki also presents staged-configuration, an incremental technique for manually determining valid feature selections. This work, however, cannot be directly applied to the configuration of large-scale DRE system configuration because it does not consider resource constraints. Also, since staged-configuration is not automated, it would take a prohibitive amount of time to determine valid system configurations.

Benavides et al. introduce the extended feature model, an augmented feature model with the ability to more articulately define features and provide additional constraints [23]. Additional descriptive information, called attributes, can be added to define one or more parameters of each feature. For example, the resource consumption and cost of a feature could be defined by adding attributes for each feature. Each of these attributes would list the type of resource and the amount consumed or provided. Additional constraints can be defined by adding extra-functional features. Extra-functional features define rules that dictate the validity of sets of attributes. For example, an extra-functional feature may require that the total cost of a set of features describing components is less than that of a feature that defines the budget. Therefore any valid feature selection would satisfy the constraint that the collective cost of the components is less than the total project budget.

Benavides et al. also provide a methodology for mapping extended feature models onto Constraint Satisfaction Problems (CSPs). A CSP is a set of Boolean variables with multiple constraints that define the values that the variables can take. Attributes and extra-functional features are maintained in the mapping. As a result, solutions that satisfy all extra-functional features and basic functional constraints can be found automatically with the use of commercial CSP solvers. Further, these solvers can be configured to optimize one or more attributes, such as the minimization of cost. These techniques, however, require the examination of all potential solutions, resulting in a system configuration that is not only valid, but also optimal.

While extended feature models and the techniques for determining valid configurations by converting them to CSPs does account for resource and budget constraints, the process is not appropriate for large-scale DRE system configuration problems. The exhaustive nature of CSP solvers often requires that all potential solutions to a problem be examined. Since the number of potential system configurations is exponential in regards to the number of potential components, the solution space is far too vast for the use of exhaustive techniques as they would require a prohibitive amount of time to determine a solution.

To circumvent the unrealistic time requirements of exhaustive search algorithms, previously we have examined approximation techniques for determining valid feature selections that satisfy multiple resource constraints using Filtered Cartesian Flattening (FCF) [25]. Approximation techniques do not require the examination of all potential configurations, allowing solutions to be determined with much greater speed. While the solutions are not guaranteed to be optimal, they are often found to be optimal or extremely near optimal.

FCF converts extended feature models into Multiple-choice Multi-dimensional Knapsack Problems (MMKP). MMKP problems, as described by Akbar et al are an extension of the Knapsack Problem (KP), Multiple-Choice Knapsack Problem (MCKP) and Multi-Dimensional Knapsack Problem (MDKP) [26]. Akbar et al. provide multiple heuristic algorithms, such as I-HEU and M-HEU for rapidly determining near optimal solutions to MMKP problems.

With FCF, approximation occurs in two separate steps. First, all potential configurations are not represented in the MMKP problems. For example, if there is an exclusive-or relationship between multiple features, then only a subset of the potentially valid relationships may be included in the MMKP problem. This pruning technique is instrumental in restricting problem size so that solving techniques can be used rapidly. Second, heuristic algorithms, such as M-HEU can be used to determine a near-optimal system configuration. M-HEU is a heuristic algorithm that does not examine all potential solutions to an MMKP problem, resulting in faster solve time, thus allowing the examination of considerably larger problems. Due to these two approximation steps, FCF can be used for problems of considerably larger size compared to methods utilizing CSPs. Figure 9 shows the distribution of the number of problems solved for a model with 10,000 features.

While FCF is capable of determining valid large scale DRE system configurations, it still makes many assumptions that may not be readily known by system designers. For example, FCF requires that the project budget for purchasing hardware and the project budget for purchasing software components be known ahead of time. The best way to split the project budget between hardware and software purchases is dictated by the configuration problem being solved. For example, if all of the hardware components are cheap and provide a huge amount of resources while the software components are expensive, it would not make sense to devote half of the project budget to hardware and half to software. A better system configuration may result from devoting 1% of the budget to hardware and 99% to software.

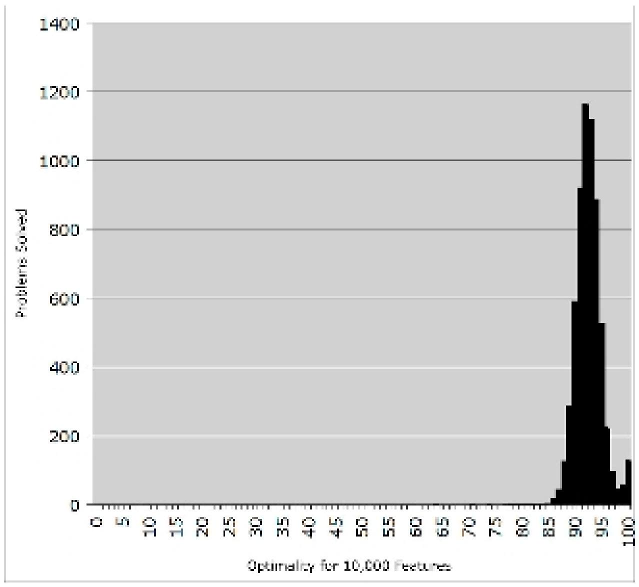


Figure 9: FCF Optimality with 10,000 Features

The Allocation baSed Configuration ExploratioN Technique (ASCENT) was developed and is capable of determining valid system configurations while also providing DRE system designers with favorable ways to divide the project budget [27]. ASCENT takes an MMKP hardware problem, MMKP software problem and a project budget amount as input. Due to the speed and performance provided by the M-HEU algorithm, ASCENT can examine many different budget allocations for the same configuration problem. ASCENT has been used for configuration problems with 1000's of features and is over 98% optimal for problems of this magnitude, making it an ideal technique for large-scale DRE system configuration.

4.4 MDE-based Configuration Modeling Methods and Tools.

This section describes meta-modeling and modeling techniques in the domain of DRE system configuration. Due to the complexities accompanying system configuration with COTS components (such as differing levels of service, complex resource interdependencies, differing resource requirements, budgetary constraints, and a multitude of component candidates), designers using manual techniques often unknowingly invalidate system configurations. MDA tools allow designers to manipulate problem entities and compare potential solutions in an environment that ensures various design rules are enforced, thereby allowing designers to focus on other problem dimensions, such as performance optimization or minimization of computational resources.

To create an MDA tool for determining DRE system configurations, we must first define several rules for DRE system configuration. First, we need to define the entities that are involved in DRE system configuration. For example, at the most basic level, DRE system configuration consists of hardware and software components. Second, we must define how these entities interact. For example, we can specify that hardware components provide computational resources and that software components consume computational resources. Finally, we need a way to define the constraints that must be maintained as these entities interact for a system configuration to be valid. For example, we may specify that a software component that interacts with a hardware component must be provided with sufficient computational resources to function by the hardware component.

This collection of rules governing the entities, interactions and constraints of the problem we are examining is defined as a Domain Specific Modeling Language, or DSML. Once we have defined a DSML, we can create model instances that enforce the rules and constraints defined by the DSML. Most nontrivial problems, however, require multiple entities, various types of interactions, and complex constraints. As a result, defining the DSML can be a confusing, arduous task. Fortunately, meta-modeling tools exist that provide a clear and simple procedure for creating Platform Independent Models (PIMs) or meta-modelings. These PIMs can then be used by modeling tools to define the DSML for created model instances, or Platform Specific Models (PSMs).

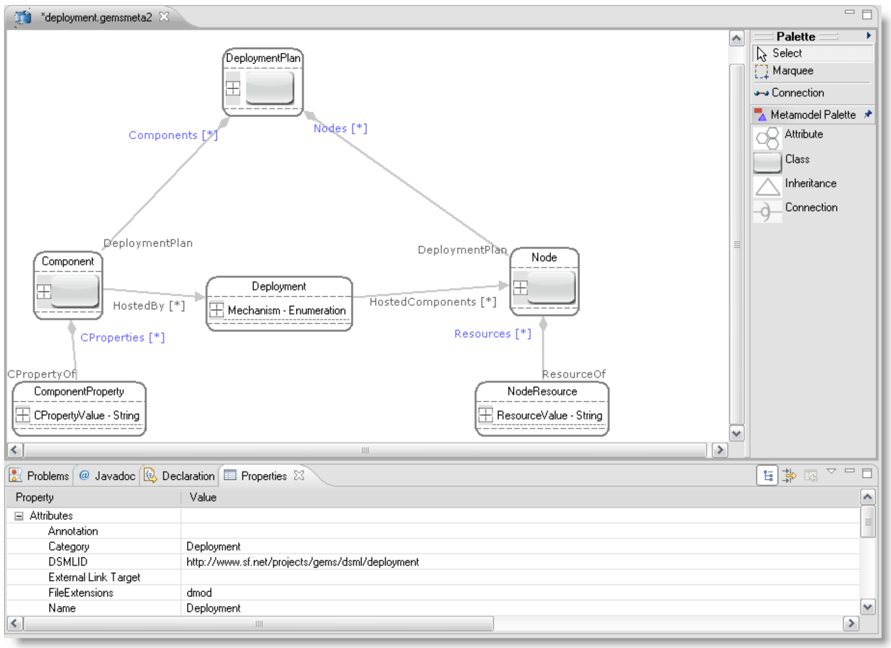
Tools for generating PIMs provide several advantages over defining DSMLs manually. First, meta-modeling DSMLs for constructing a PIM can prevent defining rules that are contradictions or inappropriate. Also, by using a meta-modeling tool for defining PIMs the DSML can easily be augmented or altered should the domain or other problem parameters change. Finally, the same complexities that are inherent to creating PSMs are also present in creating PIMs and often amplified by the additional abstraction required for creating PIMs. Meta-modeling tools use an existing DSML that defines the rules for creating meta-models, thereby enforcing the complex constraints and facilitating quick, accurate meta-model design.

Many tools exist for the definition of PIMs and creation of PSMs. Due to space constraints, we only examine the Generic Eclipse Modeling System (GEMS) [28] and the Generic Modeling

Environment (GME) [10]. GEMS is a meta-modeling tool that leverages the Eclipse Modeling Framework (EMF) and the Graphical Modeling Framework (GMF). GEMS uses a drag and drop visual interface that allows users to define meta-models without writing a single piece of code. The DSML for defining the meta-model prevents the user from making many simple mistakes, allowing the user to quickly construct a modeling tool for a specific domain. After the meta-model is constructed, the meta-model is analyzed by an interpreter. Code generators create all the necessary Java source files for defining the DSML for a domain specific modeling tool adhering to the rules and constraints defined by the meta-model.

GEMS, compared to GME, is simplified so that meta-models can be created rapidly. Figure 10 shows the DSML for describing a deployment scenario. The entities that interact within the meta-model are called *classes*. Each GEMS meta-model has a root class that serves as a container for the rest of the meta-model elements. In this case the meta-model is for a deployment plan, so the root container is called "DeploymentPlan". All classes that are a child of this root class are entities that are relevant to creating a valid deployment plan. "Node" and "Component" for example are children of the "DeplyomentPlan", as defined by the arrows leading from "Node" and "Component" to the root class. Therefore, the final modeling tool for creating deployment plans will permit the inclusion of one or more hardware "Nodes' and software "Components".

Now that we have all of the necessary entities for the modeling tool we must define the rules that govern the interactions of these entities. For example, we must define the relationship between hardware nodes and software components in which the software components consume resources of the hardware nodes. Before we can do this, however, we must define an *attribute* that specifies the resource production values of the hardware nodes and the resource consumption values of the software nodes. Once we have defined an attribute and associated it with a class, we can include the attribute in the interaction definition.



Property	Value
Attributes	
Annotation	
Category	Deployment
DSMLID	http://www.sf.net/projects/gems/dsml/deployment
External Link Target	
FileExtensions	dmod
Name	Deployment

Figure 10: Meta-modeling Defined using GEMS

An interaction is defined by adding a *connection* to the meta-modeling. The connection specifies the rules for connecting entities in the resulting PSM. For example, Figure 10 defines a "deployed on" connection between software components and hardware nodes. Within the connection, we can define additional constraints that must be satisfied for two classes to be connected. For example, for a software component to be connected to a hardware node with a "deployed on" connection the resource consumption attribute of the software component could not exceed the attribute of the hardware node that defines the amount of resource production.

Once the meta-modeling is completed, it is interpreted to create the source code for a modeling tool to enforce it. An example of the tool for modeling deployment plans is shown in Figure 11. In this figure, software components are connected to the hardware nodes on which they are deployed. If the user attempts to connect a software component to a hardware node that does not have sufficient resources for connecting the software component, the tool will not allow a connection to be made. A user can quickly create new deployment plans by dragging additional software components and hardware nodes from the toolbar. The user can then examine potential deployments simply and determine their validity by simply attempting to connect software components to hardware. Since the resource constraints and domain constraints are defined in the meta-modeling and enforced by the modeling tool, the user can create and validate deployments without having to be cognizant of these complex constraints, resulting in easier, faster deployment planning.

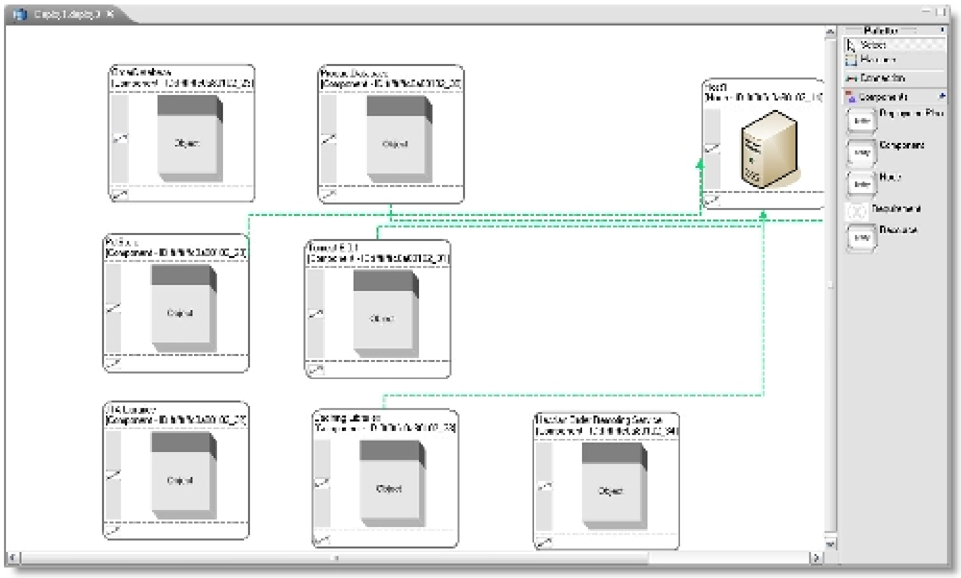


Figure 11: GEMS Deployment Plan Model

The Generic Modeling Environment (GME) [10] is another modeling toolkit for creating domain specific models. The two principal components of GME, GMeta and GModel, work together to provide this functionality. GMeta is a graphical tool for constructing meta-modelings. GMeta divides meta-modeling design into 4 separate sub-meta-modelings: the Class Diagram, Visualization, Constraints, and Attributes. The Class Diagram defines the entities within the model, known as models, atoms, and first class objects, as well as their structural hierarchy and the connections that can be made between them. The Visualization sub-meta-modeling defines different aspects, or filters, for viewing only certain entities within the model. For example, if one was defining a meta-modeling for a finite state machine, an aspect could be defined in the Visualization sub-meta-modeling that would only display accepting states in the finite state machine model. The Constraints sub-meta-modeling allows the application of Object Constraint Language (OCL) constraints to meta-modeling entities. Continuing with the finite state machine meta-modeling example, one might add a constraint that only a single starting state may exist in the model. To do this, the user would add a constraint in the Constraints sub-meta-modeling, add the appropriate OCL code to define the constraint, and then connect it to the entity to which it applies. Finally, the Attributes sub-meta-modeling allows additional data, known as attributes, to be defined and associated with other meta-modeling entities defined in the Class Diagram.

Once the meta-modeling has been constructed using GMeta, the interpreter must be run to convert the meta-modeling into a GME paradigm. This paradigm can then be loaded with GME and used to created models that adhered to the DSML defined by the meta-modeling. The user may then created models with the assurance that the design rules and domain specific constraints defined within the meta-modeling will be satisfied. Since GModel takes care of enforcing all constraints, the designer can use the graphical user interface to rapidly create and experiment with various models without the overhead of monitoring for constraint violations. If at any point the domain or design constraints of the model change, the meta-modeling can be reloaded, altered and interpreted again to change the GME paradigm appropriately. As a result, designers can easily examine scenarios in which constraints differ, giving a broader overview of the domain design space.

4.5 Case Study

In Section 4.2 we discussed the challenges of DRE system configuration. For problems of non-trivial size, these complexities proved to be too difficult to overcome without the use of programmatic techniques. In Section 4.3 we introduced several automated techniques for determining valid DRE system configurations including the Allocation baSed Configuration EsploratioN Technique (ASCENT). ASCENT is also capable of providing additional design space information, such as how to allocate a project budget, which is extremely valuable to designers.

ASCENT was implemented purely programmatically in Java. Due to this, the entire configuration problem, including external resources, constraints, software components and hardware components along with their multiple unique resource requirements had to be defined through multiple lines of complex code. As a result, the preparation time for a single configuration problem takes a considerably long amount of time. In addition, designers cannot easily manipulate many of the problem parameters to examine "what if" scenarios. The use of Model Driven Architectures, however, can circumvent these shortcomings of a purely programmatic interface. We utilized GME to construct a meta-modeling for describing a DSML for DRE system configuration and used this paradigm to experiment with the ASCENT Modeling Platform (AMP). Several benefits were observed as a result of using GME to construct an MDE for DRE system configuration as follows:

- Visual representation – Simply having a visual representation of the hardware and software components makes it significantly easier to grasp the problem, especially to users with limited experience in DRE system configuration.

- Ease of configuration – In addition to visually representing the problem, being able to quickly and easily change configuration details (budget, constraints, components, resource requirements etc.) makes the analysis much more powerful.

- Generational analysis – The model may be fed a previous solution as input, enabling designers to examine possible upgrade paths for the next budget cycle. These upgrade paths can be tracked for multiple generations, meaning that the analysis can determine the best long-term solutions. This is a capability that was not previously available with ASCENT and would have been considerably more difficult to implement without the use of GME.

- Easily Extensible- It is simple to add additional models and constraints to the existing meta-modeling. As DRE system configuration domain specific constraints are introduced, the existing meta-modeling can be altered to enforce these additional constraints in subsequent models. Since most DRE system configuration problems only slightly differ, existing meta-modelings can be reused to rapidly construct appropriate DSMLs.

- Simplifies Problem Creation – Allows user to drag and drop to create the problem instance instead of writing the 300+ required lines of complex Java code. The advantages of using a simple graphical user interface are two-fold: First, designers do not have to take the time to type such a large amount of code. Second, while typing this large amount of code designers will likely make mistakes. While many of these mistakes may be caught by the compiler, it is also likely that domain specific constraints will be inadvertently violated. Since GME enforces the design rules defined within the meta-modeling, it is not possible for the designers using GME to unknowingly make such a mistake while constructing a problem instance.

In order to expand the analytical capabilities of ASCENT, GME was utilized to provide an easily configurable, visual representation of the problem. Using these new features, it is possible to see a broader, clearer picture of the total design process as well as the global effects of even minor design decisions. We create a PIM defining a DSML for DRE system configuration using MetaGME, a Meta-modeling environment for constructing DSMLs. Meta-models created with MetaGME can be used in conjunction with GME to create models that can be interpreted using a BON2 interpreter.

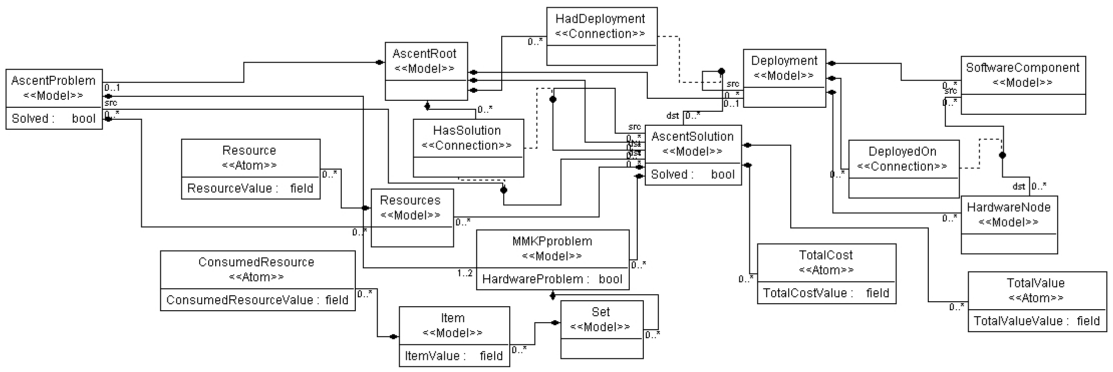


Figure 12: GME Class View Meta-modeling of ASCENT

The meta-modeling was designed and implemented using GME. Figure 12 shows the Class Diagram portion of the AMP meta-modeling. Next, a BON2 interpreter was written in C++ to handle model instances. This interpreter traverses the model and creates an XML representation of the model, which is output to a file. This XML file matches a previously defined schema for use with the Castor XML binding libraries, set of library for demarshalling XML data into Java objects. The interpreter then makes a system call to execute the ASCENTGME.jar, passing in the XML file as an argument. Within ASCENTGME.jar, several things happen. First, the XML file is demarshaled into Java objects. A class then uses these objects to create two complex MMKP Problem instances. These two problem instances, along with a total budget value, are passed to ASCENT.

ASCENT executes, and returns the best hardware and software solutions, as well as their combined cost and combined value as shown in Figure 13. Next, a First Fit Decreasing (FFD) Bin-packer uses these solutions along with their resource requirements to determine a valid deployment. This deployment data, along with the total cost, total value, hardware solution and software solution, is then written to a configuration file. The interpreter, having halted until the system call to execute the jar file terminates, then parses this configuration file. Using this data, the ASCENT solution and deployment are written back into the model. These solutions can then be examined and analyzed by designers. Designers can then change problem parameters, execute the interpreter once again, and examine the effects of the changes to the problem on the solution generated. This allows designers to rapidly examine multiple DRE system configuration design scenarios, resulting in substantially increased knowledge of the DRE system design space.

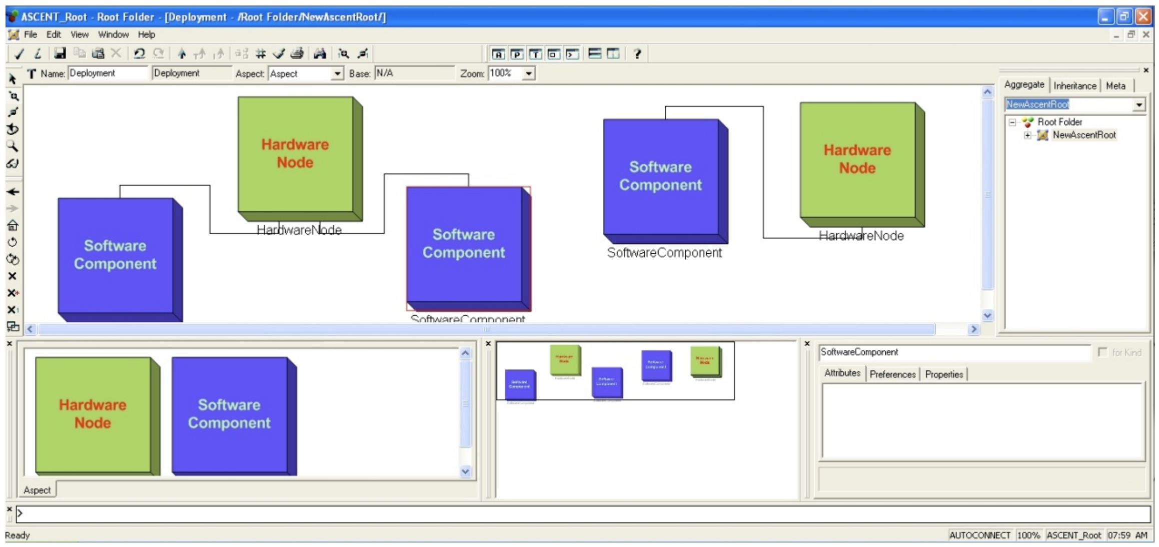


Figure 13: GME Model of Configuration Solution with AMP

4.6 Concluding Remarks

Determining valid configurations for Distributed Real-time Embedded systems is hard. Designers must take into account a myriad of constraints including resource constraints, real-time constraints, QoS constraints, and other domain specific constraints. The difficulty of these tasks is exacerbated by the presence of a plethora of potential COTS components for inclusion in the configuration, with each providing varying quality of service and functionality, and requiring different amounts of computational resources and carrying a financial cost for purchasing. The high availability of COTS components results in an exponential number of potential DRE system configurations. As a result, manual techniques for determining valid DRE system configurations are far too cumbersome. Even exact automated techniques, such as the use of Constraint Satisfaction Problems (CSPs) require a prohibitive amount of time to execute. Approximation techniques, such as ASCENT, however, do not require an exhaustive search of the vast design space allowing a much more rapid execution while often resulting in solutions with over 95% optimality.

The use of complex programmatic techniques, however, often have a steep learning curve and require large amounts of coding to construct a problem for execution. Due to the complex coding involved, these techniques carry the added burden of being error prone when defining problem instances. To combat these difficulties, we utilized an MDA-based approach that utilized GME for constructing problem instances and displaying valid solutions for DRE system configurations. The following are lessons learned during the creation of the ASCENT Modeling Platform (AMP):

- **Rapid problem construction** - Through the use of GME, problems could be constructed in a fraction of the time of using programmatic techniques.

- **Reduced errors** – Since GME uses a DSML that enforces the many complex design constraints associated with DRE system configuration, users of AMP are prevented from constructing a configuration problem that is invalid.

- **Facilitated design space exploration** – Solutions are posted directly back into the model for analysis by system designers. Designers can then change problem parameters within the model and execute the interpreter to explore multiple configuration scenarios, resulting in an increased understanding of the design space.

- **Multiple Execution Options** - Currently ASCENT is the only technique that is executed upon interpreting models. Other techniques, such as the use of CSP solvers, should be implemented to determine solutions to for problems with an appropriately reduced number of candidate components.

The current version of AMP with example code is available in open-source form at ascent-design-studio.googlecode.com.

5. Extending Model-driven Configuration Automation to Product Line Architectures

This section extends our work on model-driven configuration automation to product line architectures, which are common in DoD systems.

5.1 Introduction

Emerging trends and challenges. *Product-line architectures (PLAs)* enable the development of a group of software packages that can be retargeted for different requirement sets by leveraging common capabilities, patterns, and architectural styles. Although PLAs simplify the development of new applications by reusing existing software components, they require significant testing to ensure that valid variants function properly. Not all variants that obey the compositional rules of PLA function properly, which motivates the need for powerful testing methods and tools. For example, connecting two components with compatible interfaces can produce a non-functional variant due to assumptions made by one component, such as boundary conditions, that do not hold for the component to which it is connected.

The numerous points of variability in PLAs also yield variant configuration spaces with hundreds, thousands, or more possible variants. It is therefore crucial that PLAs undergo intelligent testing of the variant configuration space to reduce the number of configurations that must be tested. A key challenge in performing intelligent testing of the solution space is determining which variants will yield the most valuable testing results, such as performance data.

Solution approach → Model-driven testing and domain analysis of product-line architectures. Model-driven Architectures (MDA) are a development paradigm that employs models of critical system functionality, model analysis, and code generation to reduce the cost of implementing complex systems. MDA offers a potential solution to the challenges faced in testing large-scale PLAs. MDA can be used to model the complex configuration rules of a PLA, analyze the models to determine effective test strategies, and then automate test orchestration. However, effectively leveraging MDA to improve test planning and execution requires determining precisely what PLA design properties to model, how to analyze the models, and how best to leverage the results of these analyses.

5.2 PLA Modeling, Domain Analysis, and Testing Challenges

Although PLAs can increase software reuse and amortize development costs, PLA configuration spaces are hard to analyze and test manually. Deploying, configuring, and testing a PLA in numerous configurations without intelligent modeling, domain analysis, and automation is expensive and/or infeasible. Large-scale product variants may consist of thousands of component types and instances that must be tested. This large solution space presents the following key challenges to developing a PLA:

- **Challenge 1: Manually managing a PLA's configurations and constraints**. Traditional processes of identifying valid PLA variants involve software developers determining manually the software components that must be in a variant, the components that must be configured, and how the components must be composed. Such manual approaches are tedious error-prone, and do not scale well.

- **Challenge 2: Determining what PLA configurations to test through domain analysis**. With hundreds or thousands of potential configurations, testing each possible configuration may not be feasible or cost effective. Developers must determine which PLA configurations will yield the most valuable information about the capabilities of different regions of the PLA configuration space. Determining how to perform this domain analysis is hard.

- **Challenge 3: Managing the complexity of configuring, launching, and testing hundreds of valid configuration and deployment**. *Ad hoc* techniques often employ build and configuration tools, such as Make and Another Neat Tool (ANT), but application developers still must manage the large number of scripts required to perform the component installations, launch tests, and report results. Developing custom deployment and configuration scripts for each variant leads to a significant amount of reinvention and rediscovery of common deployment and configuration processes. As the number of valid variants increases, there is a corresponding rise in the complexity of developing and maintaining each variant's deployment, configuration, and testing infrastructure. Automated techniques can be used to manage this complexity.

- **Challenge 4: Evolving deployment, configuration, and testing processes as a PLA evolves**. A viable PLA must evolve as the domain changes, which presents significant challenges to the maintenance of configuration, deployment, and testing processes. Small modifications to composition rules can ripple through the PLA, causing widespread changes in the deployment, configuration, and testing scripts. Maintaining and validating the large configuration and deployment infrastructure is hard.

5.3 Model-driven Testing and Domain Analysis Techniques for Product-line Architectures

This section introduces modeling techniques for capturing PLA configuration rules and then describes how these models can be annotated with results from testing. It also presents constraint-based optimization techniques that can be used to analyze the model to derive configurations to test.

3.1 Test Automation from Feature Models.

To address Challenge 3, which is the complexity of deploying, configuring, and testing a PLA, we have developed FireAnt [29]. FireAnt is an MDA tool that allows application developers to describe the components that form the common building blocks of their PLA and to construct feature models specifying how the blocks can be composed to form valid variants. FireAnt significantly reduces the cost of testing a PLA in the following key ways:

- **Test, Deployment, and Configuration Infrastructure Generation.** FireAnt allows developers to describe the target hardware where variants will be deployed. Using a target hardware definition and the artifact mapping, FireAnt can automatically package all the archive files required to deploy each variant, as well as generate the required configuration scripts. These scripts may be implemented in a variety of languages. Currently, FireAnt provides bindings for generating ANT build files.

- **Test Automation.** FireAnt can use CSP configuration derivation techniques to generate a global configuration script that remotely deploys, configures, and tests variants automatically on each possible hardware target.

FireAnt was developed using the Generic Eclipse Modeling System (GEMS) [28], which is an open-source MDA environment built as an Eclipse plug-in. A GEMS-based meta-modeling describing the domain of PLA deployment, configuration, and testing was constructed and interpreted to create the FireAnt domain-specific modeling language (DSML) for PLAs. FireAnt's modeling environment uses GEM's support for multiple views to capture the feature model, deployment, configuration, and testing requirements of a PLA. The remainder of this section discusses how each of these views can be used to manage the complexity of testing a PLA and how the view addresses each of the challenges described in Section 5.2.

5.3.1 FireAnt Feature Modeling

To facilitate the analysis of the variant solution space and address Challenge 1 requires a formal grammar to describe the structure of the PLA and its valid configurations. This customization grammar can then be used to automatically generate and explore the variant solution space using CSP techniques.

To capture a formal definition of the PLA, the components on which it is based must be modeled. The *Feature* element is the basic building block in the Logical Composition View. A Feature represents an indivisible unit of functionality, such as an EJB or CORBA component. A configuration is a valid composition of Features that produces a complete set of application functionality. Each configuration may require different source artifacts depending on the features that it contains.

The feature model rules are specified through composition predicates. FireAnt supports that standard feature modeling constraints for AND, Exclusive OR and optional features. The children of each feature are connected through a composition predicate to their parent to specify the rules governing their selection. By capturing PLA compositional variability in a feature model through the Logical Composition tree, developers can formally specify how valid variants are composed. With a formal specification of the variant construction rules, FireAnt can automatically explore the variant solution space to discover all valid compositional variants of the PLA.

5.3.2 Dependency and Deployment Views

Simply capturing the configuration rules for the PLA is not sufficient to automate deployment and testing. FireAnt must have a specification of how the features in the feature model map down to individual source artifacts. For example, if the ConstraintsOptimizationModule is selected, what Java jar files need to be packaged into the final variant that is tested?

To automate the packaging and configuration of variants and address Challenge 3, a dependency model must be developed to associate each feature with physical artifacts, such as jar files, it relies on. This mapping from physical artifacts to PLA components can be used to automatically manage and package the artifacts and configuration scripts required for each variant. The dependency model is a platform-specific model in the MDA paradigm.

In distributed applications, developers may need to test the deployment of the application across different numbers and configurations of hardware. FireAnt's *Physical Deployment View* allows developers to specify rules on how features and their associated artifacts can be mapped to a series of remote hosts. Because the physical deployment view is not tied to any specific hardware or software implementation, it is a platform-independent model. FireAnt then takes each of these possible deployment variants and determines the unique packaging combinations of artifacts that are required for all possible valid deployments. Each unique package is called an *egg*.

The *Physical Composition View* shows which physical artifacts are associated with each egg. Individual zip archives can be created for each deployment package by traversing the Physical Composition View trees. This view manages the complexity of determining what physical artifacts should be present in for the deployment of each variant's features to a host. FireAnt can automatically collect and zip all of the required artifacts for a variant's Assemblies by traversing the Physical Composition Tree.

5.4 Results Summary

FireAnt uses the techniques described in Section 5.3 to automate (1) the generation of deployment scripts for variants, (2) the packaging of artifacts for variants, and (3) the testing of variants. These capabilities reduce the upfront cost, *A,* and enable rigorous testing of PLAs. They also address each of the four key challenges outlined in Section 5.2.

Due to the large number of variants it becomes costly for PLA developers to manually find and manage all possible variants without MDA tool support. This complexity increases the initial cost, *A*, of developing a PLA testing infrastructure since a developer must find all valid variants and determine which tests are required to ensure the proper functioning of each. In other words, $A \geq Dv + Ov$, where D_v is the time required to find each valid variant and O_v is the time required to generate an orchestration script for each variant that will execute the proper tests.

FireAnt reduces the initial cost *A* by automatically exploring the solution space and producing visualizations of valid variants for the developer. These capabilities significantly aid developer understanding of PLA variability and enables for the automated testing and packaging of each variant. Without automating the identification of variants of the PLA to test, it is hard to ensure that the PLA is tested properly, which is important in mission-critical domains.

5.5 Concluding Remarks

Product-line architectures (PLAs) can significantly improve the reuse of software components and decrease the cost of developing applications. The large number of valid variations in a PLA must be tested to ensure that only working configurations are used. Due to the large solution spaces it is infeasible or overly costly to use traditional *ad hoc* methods to test a PLA's variants.

By using MDA tools to capture the compositional and deployment variability in PLAs, we showed that much of the deployment, configuration, and testing of PLAs can be automated. This automation frees developers to focus on implementing reusable components and deployment and configuration scripts for known working units of functionality. Our experiments have shown that FireAnt can significantly reduce both the initial cost, *A,* of developing a PLA and the testing cost T_1 of each variant. FireAnt accomplishes this cost reduction by automating tedious and error-prone manual tasks, such as solution space exploration.

The following are our lessons learned from developing FireAnt and applying it to the EJB-based *Constraints Optimization System (CONST)* that schedules pickup requests to vehicles:

- **There is a larger up-front cost to adopt an automated test platform.** Initially, the cost of developing models for the MDA testing process increases development cost. Over time, however, this startup cost is amortized across variants of the SPL saving time and money.

- **Choosing the right statistical analysis technique for test results is an important concern**. This report introduces a few statistical analyses that can be used to populate quality attribute values from test results. There is a wide array of other types of analyses that can be used as well.

- **There may be unanticipated problems caused by the composition of two or more features** that may not be scriptable by FireAnt. For example, complex changes in source code may be needed. More work is needed to identify how to automate the generation of the deployment and configuration glue of PLA variants.

- **Deployment variations greatly expand the solution space** since each variant must be tested with each deployment variation. It is thus important to only model realistic deployment scenarios to restrict this space.

In future work, we are pursuing the use of FireAnt to create self-tuning installations. Many high-performance parallel computing applications, such as the Automatically Tuned Linear Algebra Software (ATLAS), run performance tests in multiple configurations as part of the installation process. These applications can then interpret the performance results to optimize themselves for the given hardware.

We also plan to expand on the ATLAS approach by allowing FireAnt users to define a fitness function based on the performance metrics collected from the individual component tests. The FireAnt test automation framework will then be used to iteratively deploy variants in various configurations in an attempt to maximize this fitness function.

Developers only need to create the tests to collect the appropriate data, such as service rate, and then provide the logic to perform analyses on the results, such as throughput analysis using queuing networks, to score the configurations. FireAnt will use this cost function to automatically deploy, configure, test, and score each candidate variant in each valid component to hardware configuration. After all testing completes, FireAnt will collect the results and install the variant/component to hardware configuration with the highest score.

6. Minimizing Number of Processors using BLITZ

This section focuses on an important topic for resource-constrained DoD-centric systems for which it is important to minimize the number of resources used to deploy DoD systems.

6.1 Introduction

Software engineers who develop distributed real-time and embedded (DRE) systems must carefully map software components to hardware. These software components must adhere to complex constraints, such as real-time scheduling deadlines and memory limitations that are hard to manage when planning deployments that map the software components to hardware [30]. How software engineers choose to map software to hardware has a direct impact on the number of processors required to implement a system.

Ideally, software components for DRE systems should be deployed on as few processors as possible. Each additional processor used by a deployment adds size, weight, power consumption, and cost to the system [31]. For example, it has been estimated that each additional pound of computing infrastructure on a commercial aircraft results in a yearly loss of $100 per aircraft in fuel costs. Likewise, each pound of processor(s) requires four additional pounds of cooling, power supply, and other support hardware. Naturally, reducing fuel consumption also reduces emissions, benefiting the environment [32].

Several types of constraints must be considered when determining a valid *deployment plan*, which allocates software components to processors. First, software components deployed on each processor must not require more resources, such as memory, than the processor provides. Second, some components may have co-location constraints, requiring that one component be placed on the same processor as another component. Moreover, all components on a processor must be schedulable to assure they meet critical deadlines [33].

Existing automated deployment techniques [34–36] leveraged by software engineers do not handle all these constraints simultaneously. For example, Rate Monotonic First-Fit Scheduling [35] can guarantee real-time scheduling constraints, but does not guarantee memory constraints or allow for forced co-location of components. Co-location of components is a critical requirement in many DRE systems. Moreover, if deploying a set of components on a processor results in CPU over-utilization, critical tasks performed by a software component may not complete by their deadline, which may be catastrophic. DRE software engineers must therefore identify deployments that meet these myriad constraints *and* minimize the total number of processors [37].

This section provides three contributions to the study of software component deployment optimizations for DRE systems that address the challenges outlined above. First, we present the *Bin packing LocatIon Technique for processor minimiZation* (BLITZ), which uses bin packing to allocate software applications to a minimal number of processors and ensure that real-time scheduling, resource, and co-location constraints are simultaneously met. Second, we present a case study that motivates the minimization of processors in a production avionics DRE system. Third, we present empirical comparisons of minimizing processors for deployments with BLITZ for three different scheduling heuristics versus the simple bin-packing of one component per processor used in the avionics case study.

6.2 Challenges of Component Deployment Minimization

This section summarizes the challenges of determining a software component deployment that minimizes the number of processors in a DRE system.

- **Rate-monotonic scheduling constraints**. To create a valid deployment, the mapping of software components to processors must guarantee that none of the software components' tasks misses its deadline. Even if rate monotonic scheduling is used, a series of components that collectively utilizes less than 100% of a processor may not be schedulable. It has been shown that determining a deployment of multiple software components to multiple processors that will always meet real-time scheduling constraints is NP-Hard [36].
- **Task co-location constraints**. In some cases, software components must be co-located on the same processor. For example, variable latency of communication between two components on separate processors may prevent real-time constraints from being honored. As a result, some components my require co-location on the same processor, which precludes the use of bin-packing algorithms that treat each software component to deploy as a separate entity.
- **Resource constraints**. To create a validate deployment, each processor must provide the resources (such as memory) necessary for the set of software components it supports to function. Developers must ensure that components deployed to a processor do not consume more resources than are present. If each processor does not provide a sufficient amount of these resources to support all tasks on the processor, a task will not be able to completely execute, resulting in a failure.

6.3 Deployment Optimization with BLITZ

The Binpacking LocatIon Technique for processor minimiZation (BLITZ) is a first-fit decreasing bin packing algorithm we developed to (1) assign processor utilization values that ensure schedulability is not exceeded and (2) enhance existing techniques by ensuring that multiple resource and co-location constraints are simultaneously honored.

6.3.1 BLITZ Binpacking

The goal of a bin packer is to place a set of items into a minimal set of bins. Each item takes up a certain amount of space and each bin has a limited amount of space available for packing. An item can be placed in a bin as long as its placement does not exceed the remaining space in the bin. Multi-dimensional bin packing extends the algorithm by adding extra dimensions to bins and items (*e.g.*, length, width, and height) to account for additional requirements of items. For example, an item may have height corresponding to its CPU utilization and width corresponding to consumed memory.

BLITZ uses an enhanced multi-dimensional bin packing algorithm to generate valid deployments that honor multiple resource constraints and co-location constraints as well as the standard real-time scheduling constraints. In BLITZ, each processor is modeled as a bin and each independent component or co-located group of components is modeled as an item. Each bin has a dimension corresponding to the available CPU utilization. Each item has a dimension that represents the CPU utilization it requires, as well as a dimension corresponding to each resource, such as memory, that it consumes. Each bin's size dimension corresponding to available CPU utilization is initialized 100%. The resource dimensions are set to the amount of each resource that the processor offers.

To pack the items, they are first sorted in decreasing order of utilization. Next, BLITZ attempts to place the first item in the first bin. If the placement of the item does not exceed the size of the bin (available resources and utilization) of the bin (processor), the item is placed in the bin. The dimensions of the items are then subtracted from the dimensions of the bin to reflect the addition of the item. If the item does not fit, BLITZ attempts to insert the item into the next bin. This step is repeated until all items are packed into bins or no bin exists that can contain the item.

Burchard et al. [36] describe several techniques that use component partitioning and bin-packing to reduce total required processors. This work, however, does not account for additional resource constraints, such as memory. Furthermore, these techniques do not allow for collocation constraints that require specific components to reside on the same processor.

6.3.2 Utilization Bounds

Conventional bin-packing algorithms assume that each bin has a static series of dimensions corresponding to available resources. For example, the amount of RAM provided by the processor is constant. Applying conventional bin-packing algorithms to software component deployment is challenge since it is hard to set a static bin dimension that guarantees the components are schedulable. Scheduling can only be modeled with a constant bin dimension of utilization if a worst-case scheduling of the system is assumed. Liu and Layland [38] have shown that a fixed bin dimension of 69.4% will guarantee schedulability, but in many cases tasks can have a higher utilization and still be schedulable.

The Liu-Layland equation states that the maximum processor utilization that guarantees schedulability is equal to $2^{1/x}-1$, where x is the total number of components allocated to the processor. With BLITZ, each bin has a scheduling dimension that is determined by the Liu-Layland equation and the number of components currently assigned to the bin. Each item will represent at least one but possibly multiple co-located components. Each time an item is assigned to a bin, BLITZ uses the Liu-Layland formula to dynamically resize the bin's scheduling dimension according to the number of components held by the items in the bin.

If the frequency of execution, or periodicity, of the components' execution requirements is known, higher processor utilization above the Liu-Layland bound is also possible. Components with harmonic periods (*e.g.*, periods that can be repeatedly doubled or halved to equal each other) can be allocated to the same processor with schedulability ensured, as long as the total utilization is less than or equal to 100%.

Unlike other deployment algorithms [36], [39], BLITZ uses multi-stage packing to exploit harmonic periods. In the first stage, components with harmonic periods are grouped together. In each successive stage, the components from the group with the largest aggregate processor utilization are deployed to the processors using a first-fit packing scheme. If not all periods of the components in a bin are harmonic, an item is allocated to a bin only if the utilization of its components fits within the dynamic scheduling Liu-Layland dimension and all other resource dimensions. If all component periods within a bin are harmonic, the utilization dimension is not dynamically calculated with Liu-Layland and a fixed value of 100% is used.

To allow for component co-location constraints, BLITZ groups components that require co-location into a single item. Each item has utilization and resource consumption equal to that of the component(s) it represents. Each item remembers the components associated with it. The Liu-Layland and harmonic calculations are performed on the individual components associated with the items in a bin and not each item as a whole.

6.4 Empirical Results

This section presents the results of applying BLITZ to an avionics case study provided by Lockheed Martin Aeronautics through the SPRUCE portal, which provides a web-accessible tool that pairs academic researchers with industry challenge problems complete with representative project data. This case study comprised 14 processors, 89 total components, and 14 co-location constraints. We compared 2 different bin-packing strategies against both BLITZ and the baseline deployment of this avionics system, produced by the original avionics domain experts.

6.4.1 Experimental Platform

All algorithms were implemented in Java and all experiments were conducted on an Apple MacbookPro with a 2.4 GHz Intel Core 2 Duo processor, 2 gigabytes of RAM, running OS X version 10.5.5, and a 1.6 Java Virtual Machine (JVM) run in client mode. All experiments required less than 1 second to complete with each algorithm.

6.4.2 Processor Minimization with Various Scheduling Bounds

This experiment compared the following bin-packing strategies against BLITZ and the baseline deployment of the avionics system: (1) a worst-case multi-dimensional bin-packing algorithm that uses 69.4% as the utilization bound for each bin, (2) a dynamic multi-dimensional bin-packing algorithm that uses the Liu-Layland equation to recalculate the utilization bound for each bin as components are added, and (3) our BLITZ technique that combines dynamic utilization bound recalculation with the harmonic period multi-stage packing. We used each technique to generate a deployment plan for the avionics system described in Section 6.2.

Figure 14 shows the original avionics system deployment, as well as deployment plans generated by the worst-case bin-packing algorithm, dynamic bin-packing algorithm, and BLITZ. The BLITZ technique required 6 fewer processors than the original deployment plan, 3 fewer processors than the worst-case bin-packing algorithm, and 1 fewer processor than the dynamic bin-packing algorithm.

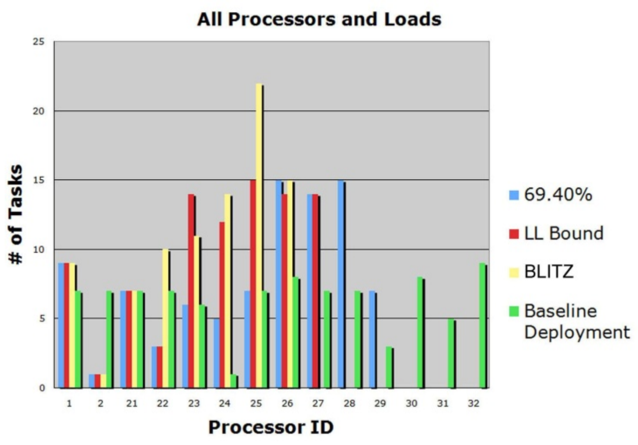


Figure 14: Deployment Plan Comparison

Figure 15 shows the total reduction of processors from the original deployment plan for each algorithm. The deployment plan generated by the worst-case bin-packing algorithm reduces the required number of processors by 3 or 21%. The dynamic bin-packing algorithm yields a deployment plan that reduces the number of required processors by 5, or 36%. BLITZ reduces the number of required processors even further, generating a deployment plan that requires 6 fewer processors, a 43% reduction.

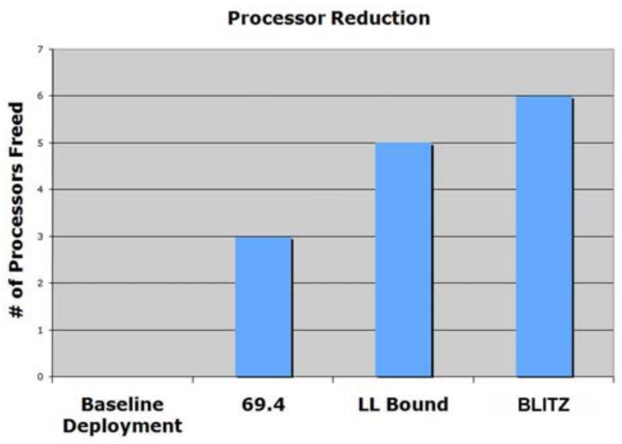


Figure 15: Scheduling Bound versus Number of Processors Reduced

6.5 Concluding Remarks

Determining component deployments that minimize the number of required processors is hard. This problem is exacerbated by proving that software applications are schedulable for a chosen deployment. Using bin packing algorithms, such as first-fit decreasing, the entire deployment space need not be searched. By using our BLITZ algorithm (which combines first-fit decreasing bin packing with proven utilization bounds based on data characteristics), valid and near minimal deployments can be determined. Based on our work with BLITZ thus far, we learned the following lessons pertaining to deployment for DRE systems:

- **Grouping based on harmonic periods improves packing tightness**. BLITZ combines the Liu-Layland equation with the increased utilization bound of components with harmonic execution periods to maximize the utilization of each processor during deployment. As a result, tasks can be clustered on fewer processors, reducing the processors required.
- **Processor minimization depends on real-time benchmarks**. BLITZ has been shown to greatly reduce the number of required processors in a DRE system of an extensively benchmarked real-time system. Without knowledge of periodicity, resource constraints, and co-location constraints, BLITZ cannot be fully utilized. It is essential to develop tools that effectively simulate and thoroughly benchmark DRE systems before they are deployed so that the full capabilities of BLITZ can be applied.

The current version of BLITZ with example code is available in open-source form at `ascent-design-studio.googlecode.com`. The industry challenge problem that is the basis for this report can be found at www.sprucecommunity.org.

7. Minimizing Network Bandwidth Usage with NOMAD

This section describes work that extends our approach to deployment optimizations by focusing on minimizing the use of network resources.

7.1 Introduction

Distributed Real-time Embedded (DRE) system designers create system deployments by mapping multiple software components to hardware. These deployments must satisfy multiple requirements, such as real-time constraints, resource constraints, collocation constraints, schedulability constraints, and budgetary constraints. Due to quantity of these constraints, creating a mapping of software components to hardware, known as a deployment plan, is extremely difficult.

Multiple DRE system deployments may exist that satisfy these constraints but vary in performance provided and resources required. Therefore, designers must also strive to determine a deployment that maximizes system performance while minimizing other system attributes, such as network bandwidth usage, power consumption, computational resource requirements such as memory, and/or financial cost. For example, a system that has a minimal financial cost and provides identical or marginally decreased performance in comparison to another extremely expensive system could be considered vastly superior.

DRE systems are subject to several constraints. Real-time constraints require that software execution complete within a set amount of time without exceeding predefined deadlines. For software to execute in a predictable manner, a scheduling of when each software task will utilize the processor must be determined. The ability to create this schedule for a set of software tasks is called schedulability and must be guaranteed for a deployment to be valid.

Collocation constraints may exist between multiple software components that require or prohibit their placement on a common processor. For example, fault tolerance concerns may require that two software tasks that provide the same functionality are not allocated to the same processor. Budgetary constraints define the total financial cost of constructing a deployment that cannot be exceeded. Therefore, the combined purchased cost of all hardware and other infrastructure, such as power, must not exceed the project budget.

Many DRE systems are mobile or deployed to remote environments without access to centralized resources, such as a power grid. Therefore, it is critical that resource consumption, such as power consumption, is minimized. For example, an Unmanned Aerial Vehicle (UAV) may fly hundreds of miles without having access to a centralized power source. To circumvent this limitation, UAVs draw power from on-board batteries that are capable of supplying a finite amount of power.

While adding batteries to a deployment increases the power availability, it also increases the weight and cost of the deployment, resulting in slower, more expensive UAVs that could potentially not fly as far, carry as much equipment, or loiter as long as the original deployment without additional batteries. Therefore, it is critical to investigate deployments that reduce the resource requirements of a system while maintaining system performance.

Design teams typically construct DRE system deployments in a distributed manner. For example, a few members of the design team may focus on satisfying schedulability constraints, while another group examines satisfying resource constraints. This allows members with heightened expertise in a specific technical area to focus on a subset of the problem without being hindered by being forced to examine other areas.

Due to decentralized design, it is not always clear what repercussions the design decisions of one group will have on another. Exacerbating this problem, the design groups may use different constraint specific models, such as Ptolemy, RT-Maude, Excel, or UML, to model design constraints and determine potential deployments. Once each group has determined a solution, considerable time and effort must be spent combining the solutions into a single representation and verifying that the deployment as a whole meets all constraints.

When constructing a DRE system deployment, system designers must determine how to assign the software required by the system to the hardware of the system for execution. The total software of the system can be split into individual units of software, called software tasks, which provide additional functionality to the final system. These software tasks require a discrete amount of computational resources, such as memory, processor utilization, and network bandwidth. The amounts of these resources required vary between each task. For the software tasks to execute, they must be allocated to a processor that possesses the necessary amount of resources.

Each additional processor used in the deployment increases the cost and power consumption of the system. Therefore, it is essential to minimize the number of processors required by the deployment. Fortunately, multiple software tasks can be assigned to a processor. Since the resource usage, real-time requirements, schedulability, and collocation constraints of each task differ, they cannot be arbitrarily assigned to processors. Optimally, the software tasks should be allocated in such a way that all design constraints are met while also minimizing the number of processors required and reducing system cost.

To do this we created BLITZ explained in Section 6 to minimize the number of processors required by a DRE system deployment. This technique, however, treats software tasks as stand-alone entities that cannot interact with one another unless collocated on a common processor. Due to the availability of a network between processors, software tasks are capable of communicating with one another regardless of their allocation. Certain software tasks must be able to transmit to other software tasks for either of them to function.

However, the use of a network for transmitting information between software tasks has several drawbacks. First, the latency of transmissions between processors differs based on the amount of traffic in the link that connects them and the topography of the network, which can make schedulability and real-time constraints more difficult to articulate and satisfy. Second, each communication between software tasks on separate processors requires additional power to transmit and receive data. Finally, software tasks that share the same processor can completely bypass the network to communicate, resulting in faster, more reliable communication that requires considerably less power by minimizing network bandwidth consumption.

Solution Approach→Collocation of Network Intensive Software Tasks: Here we present a methodology for intelligently allocating software tasks to processors in such a way that ensures all design constraints are met while reducing total network bandwidth consumption. Placing software tasks that require large amounts of bandwidth to communicate on a common processor reduces the need for network transmissions. The software tasks can use the hardware on the processor to communicate without utilizing the network, resulting in decreased network bandwidth and power consumption.

This section also provides four contributions to the area of DRE system deployment. First, we present the NetwOrk MinimizAtion Depolyment (NOMAD) technique for determining system deployments that satisfy DRE system constraints while minimizing network bandwidth consumption. Second, we present a case study of the application of NOMAD to minimize the network bandwidth of a DRE aeronautics system. Third, we present the New Associative Object Model of Integration (NAOMI), a multi-model manager tool for handling different design constraint representations. Last, we compare our empirical results with an existing aeronautics system to demonstrate the potential reduction of network bandwidth consumption.

7.2 Challenges in Deployment Network Bandwidth Minimization

Constructing a system deployment plan consists of assigning software tasks to processors for execution. Several constraints, such as real-time constraints, fault tolerance constraints, budgetary constraints, resource constraints, and collocation constraints must be satisfied. If one or more of these constraints are violated, the system deployment will exhibit unpredictable behavior potentially causing catastrophic system failures.

Several valid deployments may exist that satisfy all required constraints. These deployments, however, require different amounts of power and hardware to function and carry different financial costs. Since resource availability and project budgets are normally limited, a deployment that requires less power, hardware, or money to deploy is certainly favorable to a more expensive deployment that requires additional resources. Therefore, it is essential to determine a deployment that not only satisfies all constraints, but also minimizes resource requirements and deployment cost.

7.2.1 Challenge 1: Tolerating Processor Failure

DRE systems must remain functional in the inevitable event of system failures. These failures could be the result of poorly constructed hardware, harsh environmental conditions, or even as a result of enemy attacks. The amount of failures that a system must be able to sustain is unique to the domain and application of the deployment. The primary method for increasing the fault tolerance capabilities of a system deployment is to create replicas.

A replica is an additional node that is deployed with the same software tasks as an existing node. In the event that the original node fails for any reason, the replica can continue to provide the functionality that the failed node provided. Replicating existing software allocations onto additional processors increases the fault tolerance of a system deployment. However, there are several disadvantages to deploying a large quantity of replicas. Each replica requires the purchase of additional hardware, resulting in increased system cost, power consumption, weight, and heat generation.

7.2.2 Challenge 2: Collocating Software Tasks

Real-time constraints may require that two or more software tasks be allocated to the same processor. For example, two software tasks may require the transmission of large amounts of data between each other. If the software tasks are allocated to separate processors, then data must be transmitted over the network. As a result of network latency, the software tasks would take additional time to function. This could lead to missed deadlines and the violation of real-time constraints. Further, if two software tasks require each other to function and are placed on separate processors, then the failure of either processor would prevent both software tasks from executing to completion.

The collocation of these two tasks on the same processor allows them to communicate with each other without utilizing the network, resulting in reduced communication latency. Also, the collocation of two software tasks that depend on each other to function reduces the number of processor failures that would disallow the execution of the tasks.

7.2.3 Challenge 3: Minimizing Resource Requirements

DRE systems require various amounts of multiple resources, such as memory and power, to function. As the computational requirements of the software tasks included in the deployment increase, additional resources must be purchased and included in the system deployment. Further, additional power must be supplied to meet power consumption requirements as the number of processors in the deployment increases. However, it is not always simple or even possible to increase the power production of a DRE system.

For example, Unmanned Aerial Vehicles (UAVs) must meet stringent weight requirements. As the weight of the UAV increases, the distance that the aircraft can fly, the amount of time it can loiter, and the size of the payload that it can deliver to a target decreases, reducing its effectiveness. Since UAVs are mobile crafts, batteries must be utilized to produce the necessary power. These batteries, however, are extremely heavy and expensive. Therefore, reducing the power requirements of the system can reduce the size and number of batteries need on the aircraft, resulting in a cheaper, more effective UAV.

7.2.4 Challenge 4: Meeting Budgetary Constraints

Traditionally, hardware components, such as processors, and software applications must be purchased before they can be included in a system deployment. As deployments increase in size, the computational resource requirements increase, requiring the purchase of additional hardware. As a result of these purchases, the financial cost of the system deployment increases.

Most projects, however, have a finite project budget for constructing and deploying the system. The total cost of the system must not exceed the project budget. Further, it is extremely desirable that the financial cost of the system be minimized so that excess funds can be saved for future endeavors.

Several factors directly impact the financial cost of a system deployment. First, creating additional replicas to provide increased fault tolerance requires the purchasing of additional hardware. Second, each additional processor increases resource requirements of the deployment. Additional resources, such as batteries to provide power, must be purchased and added to the system deployment. Finally, each additional processor increases the heat generation of the system, requiring that additional heat sinks and cooling mechanisms be purchased and included in the deployment.

7.2.5 Challenge 5: Ensuring Schedulability of Software Tasks

Assigning multiple software tasks to a processor rather than allocating a single processor for each software task greatly reduces the number of processors required, resulting in reduced system cost, size, and network bandwidth consumption. However, the inclusion of multiple software tasks to a single processor requires that a schedule be created that allows all of the tasks to execute to completion prior to real-time deadlines. Otherwise, all of the software tasks may not be given enough time on the processor, resulting in unpredictable system behavior and potentially system failure. Designers must ensure that at least one valid execution schedule exists when assigning additional software tasks to a processor. This constraint, known as schedulability, must be satisfied to guarantee that all software tasks will receive enough time on the processor to execute to completion before deadlines are reached.

7.2.6 Challenge 6: Reconciling Multiple Constraint Models

The creation of a DRE system deployment requires the formalization of many constraints, including fault tolerance constraints, collocation constraints, resource constraints, and budgetary constraints. It is rare, however, that a single person defines these constraints. Normally, a small group of people tackles only a single facet of the system design and deployment. One person may focus on the fault tolerance requirements of the system while another examines resource requirements. These two people may utilize entirely different methodologies and represent their results in models of different types.

For example, one group may use a Ptolemy model for analyzing fault tolerance requirements while another person may use an Excel model for defining the budgetary restrictions of the system. The final system deployment, however, must take into account all off the constraints regardless of their representation. Therefore, techniques must be constructed that allow seamless data exchange between models of different types so that a system deployment can be constructed that meets all design constraints.

7.3 Network Bandwidth Minimization with NOMAD

The NetwOrk MinimzAtion Deployment (NOMAD) technique uses a hybrid evolutionary-heuristic algorithm that utilizes Particle Swarm Optimization (PSO) [40] and bin packing techniques to determine deployments that strive to minimize the network bandwidth consumption of a deployment while also honoring the constraints described in Section 7.2. This technique, however, requires that all design constraints, available hardware components, software tasks, and metadata describing the execution of the software tasks be available in a common format for input into the technique.

This poses a formidable obstacle since many distributed, disparate groups define these constraints with minimal communication between one another. As a result, the design constraints are often represented in different complex models. Therefore, it is difficult to extract the necessary data from these models to serve as input to NOMAD. Multi-modeling tools exist, however, such as NAOMI M3, for managing multiple model representations of data. NAOMI is utilized to facilitate data sharing and aggregation between models of different types so that dissimilarly represented constraint data can be supplied to the NOMAD algorithm. Once the design data has been collected using NAOMI, NOMAD can execute to determine a deployment that minimizes network bandwidth consumption.

NOMAD is a hybrid algorithm utilizing bin-packing and PSO techniques. First, the scheduling data of the software tasks is analyzed. This data includes the periodicity of each task and the processor utilization requirement of each task. The periodicity of a task is the set of intervals that defines when a software task must utilize the processor. The processor utilization defines what percentage of the processor must be available to the task during the interval for the task to complete before its deadline arrives.

Techniques have been studied that can ensure that a schedule exists for a set of tasks based on the periodicity data and processor utilization. For example, if the software tasks exhibit harmonic periodicity, a set of tasks can be considered schedulable if the combined processor utilization is less than 100% for any interval. The Liu-Layland bound [38], as discussed in the previous section, is an equation that uses the number of software tasks allocated to a processor to set a bound that guarantees schedulability. Finally, regardless of the number of software tasks allocated to a processor schedulability can be guaranteed if the combined utilization of the tasks does not exceed 69.4%, as proven by Liu-Layland.

First Fit Decreasing (FFD) bin-packing can be used to determine deployments that minimize network bandwidth consumption while also meeting all design constraints. The software tasks are sorted in decreasing order by the network bandwidth requirements between individual software tasks. Software components that require collocation are combined and treated as a single task. We attempt to place the first software task pair onto the first processor, represented as a bin. If the combined processor utilization is less than the appropriate scheduling bound, then the two tasks will satisfy real-time and schedulability constraints. If enough computational resources are available on the processor, then the software tasks are deployed to the processor. If they cannot fit then the two tasks will not be able to be collocated and the next task set is attempted for placement.

There is an exponential number of potential deployment plans that could be selected. Due to the NP-hard nature of this problem, it would take a prohibitive amount of time to examine all of the potential software task allocations. While FFD bin-packing usually delivers quality solutions, other solutions may exist that cannot be found by simply sorting the tasks in order of processor utilization. To discover these solutions, we use PSO to determine an initial packing of a small subset of the software tasks. Once several of the software tasks are allocated, FFD bin-packing is commenced to allocate the remaining software tasks to the processors. This allows for the discovery of solutions that would not be found by using FFD bin-packing alone.

7.4 Concluding Remarks

DRE system deployment requires that myriad design constraints be satisfied. This characteristic combined with the exponential number of potential system deployments makes determining deployments with minimal network minimization difficult. This difficulty is further compounded by the distributed nature of DRE system design across many small teams that utilize different modeling methodologies to represent design constraints. Multi-modeling mangers, such as NAOMI, can be used to mitigate the difficulties of using multiple model representations of constraints across a single project. Without tools such as NAOMI, the time and effort required aggregating and supplying design constraints for use with cutting-edge deployment techniques would be vastly increased, potentially to the point of becoming prohibitive.

NOMAD can be used to determine deployments that have drastically reduced network bandwidth. By using hybrid evolutionary-heuristic algorithms with PSO and FFD bin-packing, new valid deployments can be determined in a matter of minutes, despite the staggering size of the exponential deployment state space. As a result of network bandwidth minimization, power consumption is reduced and execution time is expedited, resulting in more effective DRE system deployments.

8. ScatterD: Multi-objective Deployment Optimization

This section describes our effort on the deployment and configuration of distributed real-time and embedded (DRE) systems focusing on multi-objective deployment optimizations, i.e., when considering more than one parameter as a constraint in contrast to only one as described in Sections 6 and 7.

8.1 Introduction

Current trends and challenges. Several trends are shaping the development of embedded avionics systems. First, there is a migration away from older *federated computing architectures* where each subsystem occupied a physically separate hardware component to *integrated computing architectures* where multiple software applications implementing different capabilities share a common set of computing platforms. Second, publish/subscribe (pub/sub) based messaging systems are increasingly replacing the use of hard-coded cyclic executions.

These trends are yielding a number of benefits. For example, integrated computing architectures create an opportunity for system-wide optimization of *deployment topologies*, which map software components and their associated tasks to hardware processors. Optimized deployment topologies can pack more software components onto the hardware, thereby optimizing system processor, memory, and I/O utilization [41–43]. Increasing hardware utilization can decrease the total hardware processors that are needed, lowering both implementation costs and maintenance complexity. Moreover, reducing the required hardware infrastructure has other positive side effects, such as reducing weight and power consumption.

Open problems. Developing computer-assisted methods and tools to deploy software to hardware in embedded systems is hard [30] [44] due to the number and complexity of constraints that must be addressed. For example, developers must ensure that each software component is provided with sufficient processing time to meet any real-time scheduling constraints [45]. Likewise, resource constraints (such as total available memory on each processor) must also be respected when mapping software components to hardware components [45][46]. Moreover, assigning real-time tasks in multiprocessor and/or single-processor machines is *NP-Hard* [36], which means that such a large number of potential deployments exist that it would take years to investigate all possible solutions.

Current algorithmic deployment techniques are largely based on heuristic bin-packing [34–36], which represents the software tasks as *items* that take up a set amount of space and hardware processors as *bins* that provide limited space. Bin-packing algorithms try to place all the items into as few bins as possible without exceeding the space provided by the bin in which they are placed. These algorithms use a heuristic, such as sorting the items based on sized and placing them in the first bin they fit in, to reduce the number of solutions that are considered and avoid exhaustive solution space exploration.

Conventional bin-packing deployment techniques take a one-dimensional view of deployment problems by just focusing on a single deployment concern at a time. Example concerns include resource constraints, scheduling constraints, or fault-tolerance constraints. In production avionics systems, however, deployments must meet combinations of these concerns simultaneously.

Solution approach -> Computer-assisted deployment optimization. This section describes and validates a method and tool called *ScatterD* that we developed to perform computer-assisted deployment optimization for avionics systems. The ScatterD model-driven engineering deployment tool implements the *Scatter Deployment Algorithm* [47], which combines heuristic bin-packing with optimization algorithms, such as genetic algorithms [48] or swarm optimization techniques [49] that use evolutionary or bird flocking behavior to perform blackbox optimization. This report shows how avionics system developers have used ScatterD to automate the reduction of processors and network bandwidth in complex embedded system deployments.

8.2 Modern Flight Embedded Avionics Systems: A Case Study

Over the past 20 years, avionics systems have become increasingly sophisticated. Modern aircraft now depend heavily on software executing atop a complex embedded network for higher-level capabilities, such as more sophisticated flight control and advanced mission computing functions. To accommodate the increased amount of software required, avionics systems have moved from older federated computing architectures to integrated computing architectures that combine multiple software applications together on a single computing platform containing many software components.

The class of avionics system targeted by our work is a networked parallel message-passing architecture containing many computing nodes. At the individual node level, ARINC 653-compliant time and space partitioning separates the software applications into sets with compatible safety and security requirements. Inside a given time partition, the applications run within a hard real-time deadline scheduler that executes the applications at a variety of harmonic periods.

Integrated computing architectures have many benefits and challenges. Key benefits include better optimization of hardware resources and increased flexibility, which result in a smaller hardware footprint, lower energy use, decreased weight, and enhanced ability to add new software to the aircraft without updating the hardware. The key challenge, however, is increased system integration complexity. In particular, while the homogeneity of processors gives system designers a great deal of freedom allocating software applications to computing nodes, optimizing this allocation involves simultaneously balancing multiple competing resource demands.

For example, even if the processor demands of a pair of applications would allow them to share a platform, their respective I/O loads may be such that worst-case arrival rates would saturate the network bandwidth flowing into a single node. This problem is complicated for single-core processors used in current integrated computing architectures. Moreover, this problem is being exacerbated with the adoption and fielding of multi-core processors, where competition for shared resources expands to include internal buses, cache memory contents, and memory access bandwidth.

8.3 Deployment Optimization Challenges

This section describes the challenges facing developers when attempting to create a deployment topology for an avionics system. The discussion below assumes a networked parallel message-passing architecture (such as the one described in Section 8.2). The goal is to minimize the number of required processors and the total network bandwidth resulting from communication between software tasks.

- **Challenge 1: Satisfying rate-monotonic scheduling constraints efficiently**. The deployment topology must ensure that the set of software components allocated to each processor are schedulable and will not miss real-time deadlines. Finding a deployment topology for a series of software components that ensures schedulability of all tasks is called "multiprocessor scheduling" and is NP-Hard [36].

- **Challenge 2: Reducing the complexity of memory, cost, and other resource constraints**. Processor execution time is not the only type of resource that must be managed while searching for a deployment topology. Hardware nodes often have other limited but critical resources, such as main memory or core cache necessary for the set of software components they support.

- **Challenge 3: Satisfying complex dynamic network resource and topology constraints**. Embedded avionics systems must often ensure that not only processor resource limitations are adhered to, but network resources (such as bandwidth) are not over consumed. Adding the consideration of network resources to deployment substantially increases the complexity of finding a software-to-hardware deployment topology mapping that meets requirements.

8.4 ScatterD: A Deployment Optimization Tool to Minimize Bandwidth and Processor Resources

Heuristic bin-packing algorithms work well for multiprocessor scheduling and resource allocation. However, heuristic bin-packing is not effective for optimizing designs for certain system-wide properties, such as network bandwidth consumption, and hardware/software cost. Below we explain how ScatterD integrates the ability of heuristic bin-packing algorithms to generate correct solutions to scheduling and resource constraints with the ability of metaheuristic algorithms to flexibly minimize network bandwidth and processor utilization and address the challenges in Section 8.3.

8.4.1 Satisfying real-time scheduling constraints with ScatterD

ScatterD ensures that the numerous deployment constraints (such as the real-time schedulability constraints described in Challenge 1 from Section 8.3) are satisfied by using heuristic bin-packing to allocate software tasks to processors. Conventional bin-packing algorithms for multiprocessor scheduling are designed to take as input a series of items (*e.g.*, tasks or software components), the set of resources consumed by each item (*e.g.*, processor and memory), and the set of bins (*e.g.*, processors) and their capacities. The algorithm outputs an assignment of items to bins (*e.g.*, a mapping of software components to processors).

ScatterD ensures schedulability of the avionics system by using response-time analysis [50]. The response time resulting from allocating a software task of the avionics system to a processor is analyzed to determine if a software component can be scheduled on a given processor before allocating its associated item to a bin. If the response time is fast enough to meet the real-time deadlines of the software task, the software task can be allocated to the processor.

8.4.2 Satisfying Resource Constraints with ScatterD

To ensure that other resource constraints (such as memory requirements described in Challenge 2 from Section 8.3) of each software task are met, we specify a capacity for each bin that is defined by the amount of each computational resource provided by the corresponding processor in the avionics hardware platform. Similarly, the resource demands of each avionics software task define the resource consumption of each item. Before an item can be placed in a bin, ScatterD verifies that the total consumption of each resource utilized by the corresponding avionics software component and software components already placed on the processor does not exceed the resources provided.

8.4.3 Minimizing Network Bandwidth and Processor Utilization with ScatterD

To address deployment optimization issues (such as those raised in Challenge 3 from Section 8.3), ScatterD uses heuristic bin-packing to ensure that schedulability and resource constraints are met. If the heuristics are not altered, the bin-packing algorithm will always yield the same solution for a given set of software tasks and processors. The number of processors utilized and the network bandwidth requirements will therefore not change from one execution of the bin-packing algorithm to another. In a vast deployment solution space associated with a large-scale avionics system, however, there may be many other deployments that substantially reduce the number of processors and network bandwidth required, while also satisfying all design constraints.

To search for avionics deployment topologies with minimal processor and bandwidth requirements—while still ensuring that other design constraints are met—ScatterD uses metaheuristic algorithms to *seed* the bin-packing algorithm. In particular, metaheuristic algorithms are used to search the deployment space and select a subset of the avionics software tasks that must be packed prior to the rest of the software tasks. By forcing an altered bin-packing order, new deployments with different bandwidth and processor requirements are generated.

Since bin-packing is still the driving force behind allocating software tasks, design constraints have a higher probability of being satisfied. By using metaheuristic algorithms to search the design space— and then using bin-packing to allocate software tasks to processors—ScatterD can generate deployments that meet all design constraints while also minimizing network bandwidth consumption and reducing the number of required processors in the avionics platform.

8.5 Results Summary

The first experiment examined applying ScatterD to minimize the number of processors in the legacy avionics system deployment, which originally consisted of software tasks deployed to 14 processors. Applying ScatterD with swarm optimization techniques and genetic algorithms resulted in increased utilization of the processors, reducing the number of processors needed to deploy the software to eight in both cases. The remaining six processors could then be removed from the deployment without affecting system performance, resulting in a 43% reduction.

The ScatterD tool was also applied to minimize the bandwidth consumed due to communication by software tasks allocated to different processors in the legacy avionics system. Reducing the bandwidth requirements of the system leads to more efficient, faster communication while also reducing power consumption.

8.6 Concluding Remarks

Optimizing deployment topologies on a legacy embedded avionics system can yield substantial benefits, such as reducing hardware costs and power consumption. The following is a summary of the lessons we learned applying our ScatterD tool for deployment optimization to a legacy avionics system:

- **Multiple constraints make deployment planning hard**. Avionics deployments must adhere to a wide range of strict constraints, such as resource, colocation, scheduling, and network bandwidth. Deployment optimization tools must account for all these constraints when determining a new deployment.

- **A huge deployment space requires intelligent search techniques.** The vast majority of potential deployments that could be created violate one or more design constraints. Intelligent and automated techniques, such as hybrid-heuristic bin-packing, should therefore be applied to discover valid "near-optimal" deployments.

- **Substantial processor and network bandwidth reductions are possible.** Applying hybrid-heuristic bin-packing to the avionics system resulted in 42.8% processor reduction and 24% bandwidth reduction. Our future work is applying hybrid-heuristic binpacking to other embedded system deployment domains, such as automobiles, multi-core processors, and tactical smart phone applications.

The ScatterD tool is available in open source form in the Ascent Design Studio (ascent-design-studio.googlecode.com). A document describing the avionics system case study, as well as additional information on ScatterD, can be found at the SPRUCE web portal (www.sprucecommunity.org), which pairs open industry challenge problems with cutting-edge methods and tools from the research community.

9. SCORCH: Model-driven auto-scaling in Cloud Platforms

This section describes how model-driven technologies can be used to enable auto-scaling in cloud platforms, which are increasingly becoming attractive environments for operational DoD systems. It also describes how the model-driven technology developed under this effort, SCORCH, is used in power management for cloud environments.

9.1 Introduction

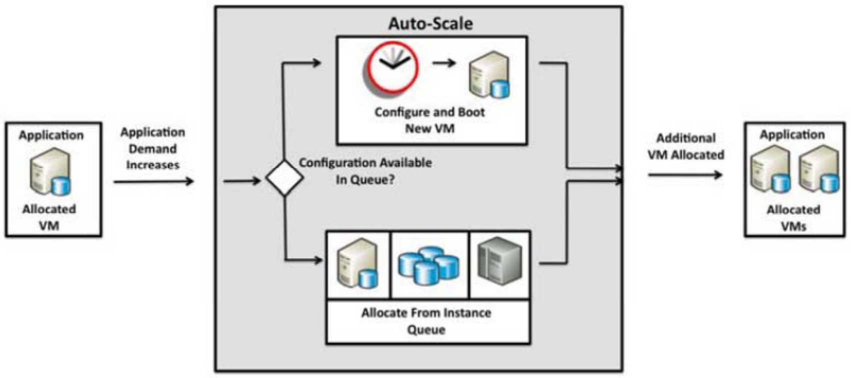
Current trends and challenges: By the end of 2011, power consumption of computing data centers is expected to exceed 10,000,000,000 kilowatt-hours (kWh) and generate over 40,568,000 tons of CO2 emissions [55–58]. Since data centers operate at only 20-30% utilization, 70-80% of this power consumption is lost due to over-provisioned idle resources resulting in roughly 29,000,000 tons of unnecessary CO2 emissions. Therefore, applying new computing paradigms, such as cloud computing with auto-scaling, to increase server utilization and decrease idle time is paramount to creating greener computing environments with reduced power consumption and emissions [59–63].

Cloud computing is a computing paradigm that uses virtualized server infrastructure and auto-scaling to dynamically provision virtual OS instances [51]. Rather than over-provisioning an application's infrastructure to meet peak load demands, an application can *auto-scale* by dynamically acquiring and releasing virtual machine instances as load fluctuates. Auto-scaling results in increased server utilization and decreased idle time in comparison to over-provisioned infrastructures in which superfluous system resources may remain idle, resulting in unnecessary power consumption and ultimately superfluous CO2 emissions. Further, by allocating virtual machines to applications on demand, cloud infrastructure users can pay for servers incrementally rather than investing the large up-front costs to purchase new servers, resulting in reduced up-front operational costs.

Although cloud computing can help reduce idle resources and negative environmental impact, running with less instantly available computing capacity can impact quality of service as load fluctuates. For example, a prime time television commercial advertising a hit new product may cause a ten-fold increase in traffic to the advertisers' website for about 15 minutes. Data centers can use existing idle resources to handle this momentary increase in demand and maintain quality of service. Without these additional resources, the quality of service of the website would degrade, resulting in an unacceptable user experience. However, if this commercial only airs twice a week, then these additional resources sit idle for the rest of the week, consuming additional power without being utilized.

Devising mechanisms for reducing power consumption and environmental impact through cloud auto-scaling is difficult. Auto-Scaling must ensure that virtual machines can be provisioned and booted quickly enough to meet response time requirements as the load changes. If auto-scaling is too slow to keep up with load fluctuations, applications may experience a period of poor response time while waiting for additional computational resources to come online. One way to mitigate this risk is to maintain an auto-scaling queue containing pre-booted and pre-configured virtual machine instances that can be allocated rapidly, as shown in Figure 16.

When a cloud application requests a new virtual machine configuration from the auto-scaling infrastructure, the auto-scaling infrastructure first attempts to fulfill the request with a pre-booted virtual machine in the queue. For example, if a virtual machine with Fedora Core 6, JBoss, and MySQL is requested, the auto-scaling infrastructure will attempt to find a matching virtual machine in the queue. If no match is found, a new virtual machine must be booted and configured to match the allocation request.



Auto-Scale
Application
Allocated VM
Application Demand Increases
Configure and Boot New VM
Configuration Available In Queue?
Allocate From Instance Queue
Additional VM Allocated
Application
Allocated VMs

Figure 16: auto-scaling in a Cloud Infrastructure

Open problems: A key challenge for developers is determining green settings for the size and properties of the auto-scaling queue shared by multiple applications that may have different virtual machine configurations [52]. The chosen configurations must meet the configuration requirements of multiple applications and reduce power consumption without adversely impacting quality of service. For example, a web application may request virtual machine instances configured as database, middle-tier Enterprise Java Beans (EJB), or front-end web servers. Determining how to capture and reason about the configurations that comprise the auto-scaling queue is hard due to the large number of configuration options (such as MySQL and SQL Server databases, Ubuntu Linux and Windows operating systems, and Apache HTTP and IIS/Asp.Net web hosts) offered by cloud infrastructure providers.

It is even harder to determine the optimal queue size and types of virtual machine configurations that will ensure that virtual machine allocation requests can be serviced quickly enough to meet a required auto-scaling response time limit. Cost optimization is challenging because each configuration placed into the queue can have varying costs based on the hardware resources and software licenses it uses. Finally, energy consumption minimization is also difficult as each set of hardware resources also consumes a different amount of power.

Solution approach ⇗ ! Auto-scaling queue configuration derivation based on feature models. In this research we explored a Model-Driven Engineering (MDE) approach called the *Smart Cloud Optimization for Resource Configuration Handling* (SCORCH), which is a continuation of our earlier effort. SCORCH captures virtual machine configuration options for a set of cloud applications and derives an optimal set of virtual machine configurations for an auto-scaling queue. From the perspective of environmental and power concerns, SCORCH provides three contributions to the green computing through MDE based cloud auto-scaling. First, we describe an MDE technique for transforming feature model representations of cloud virtual machine configuration options into constraint satisfaction problems (CSPs) [53], [54]. Second, we describe another MDE technique for analyzing application configuration requirements, virtual machine power consumption, and operating costs to determine what virtual machine instance configurations to include in an auto-scaling queue in order to meet an auto-scaling response time guarantee while minimizing power consumption. Third, we present empirical results from a case study using Amazon's EC2 cloud computing infrastructure that shows our MDE techniques minimize power consumption and operating cost while ensuring that an auto-scaling response time requirement is met.

9.2 Challenges of Configuring Virtual Machines in Cloud Environments

Reducing unnecessary idle system resources by applying auto-scaling queues can potentially lead to vast reductions in power consumption and resulting *CO*2 emissions. However, quality of service demands, various configuration requirements and other challenges make achieving a greener computing environment difficult. This section describes three key challenges of capturing virtual machine configuration options and using this configuration information to optimize the setup of an auto-scaling queue to minimize power consumption. Section 9.3 then presents SCORCH's MDE approach to resolving these challenges.

9.2.1 Challenge 1: Capturing Virtual Machine Configuration Options and Constraints

Cloud computing can lead to greener computing by reducing power consumption. A cloud application can request virtual machines with a wide range of configuration options, such as type of processor, amount of memory, OS, and installed middleware, all of which consume different amounts of power. For example, the Amazon EC2 cloud infrastructure supports 5 different types of processors, 6 different memory configuration options, and over 9 different OS types, as well as multiple versions of each OS type [64]. The power consumptions of these configurations range from 150 to 610 Watts per hour. These EC2 configuration options cannot be selected arbitrarily and must adhere to a multitude of configuration rules. For example, a virtual machine running on Fedora Core 6 OS cannot run MS SQL Server. Tracking these numerous configuration options and constraints is hard.

9.2.2 Challenge 2: Selecting Virtual Machine Configurations to Guarantee Auto-scaling Speed Requirements

While reducing idle resources results in less power consumption and greener computing environments, cloud computing applications must also meet quality of service demands. A key determinant of auto-scaling performance is the types of virtual machine configurations that are kept ready to run. If an application requests a virtual machine configuration and an exact match is available in the auto-scaling queue, the request can be fulfilled nearly instantaneously. If the queue does not have an exact match, it may have a running virtual machine configuration that can be modified to meet the requested configuration faster than provisioning and booting a virtual machine from scratch. For example, a configuration may reside in the queue that has the correct OS but needs to unzip a custom software package, such as a pre-configured Java Tomcat Web Application Server, from a shared file system onto the virtual machine. Auto-scaling requests can thus be fulfilled with both exact configuration matches and subset configurations that can be modified faster than provisioning a virtual machine from scratch.

Determining what types of configurations to keep in the auto-scaling queue to ensure that virtual machine allocation requests are serviced fast enough to meet a hard allocation time constraint is hard. For one set of applications, the best strategy may be to fill the queue with a common generic configuration that can be adapted quickly to satisfy requests from each application. For another set of applications, it may be faster to fill the queue with the virtual machine configurations that take the longest to provision from scratch. Numerous strategies and combinations of strategies are possible, making it hard to select configurations to fill the queue that will meet auto-scaling response time requirements.

9.2.3 Challenge 3: Optimizing Queue Size and Configurations to Minimize Energy Consumption and Operating Cost

A further challenge for developers is determining how to configure the auto-scaling queue to minimize the energy consumption and costs required to maintain it. The larger the queue, the greater is the energy consumption and operating cost. Moreover, each individual configuration within the queue varies in energy consumption and cost. For example, a “small” Amazon EC2 virtual machine instance running a Linux-based OS consumes 150W and costs $0.085 per hour while a "Quadruple Extra Large" virtual machine instance with Windows consumes 630W and costs $2.88 per hour. It is hard for developers to manually navigate the tradeoffs between energy consumption, operating costs and auto-scaling response time of different queue sizes and sets of virtual machine configurations. Moreover, there are an exponential number of possible queue sizes and configuration options that complicates deriving the minimal power consumption/operating cost queue configuration that will meet auto-scaling speed requirements.

9.3 SCORCH: An MDE Based Optimization Technique for Generating Cloud Auto-Scaling Queues

This section describes the MDE techniques that SCORCH uses to address the challenges of optimizing an auto-scaling queue described in Section 9.2. SCORCH resolves these challenges by using models to capture virtual machine configuration options explicitly, model transformations to convert these models into constraint satisfaction problems (CSPs), and constraint solvers to derive the optimal queue size and contained virtual machine configuration options to minimize cost while meeting auto-scaling response time requirements.

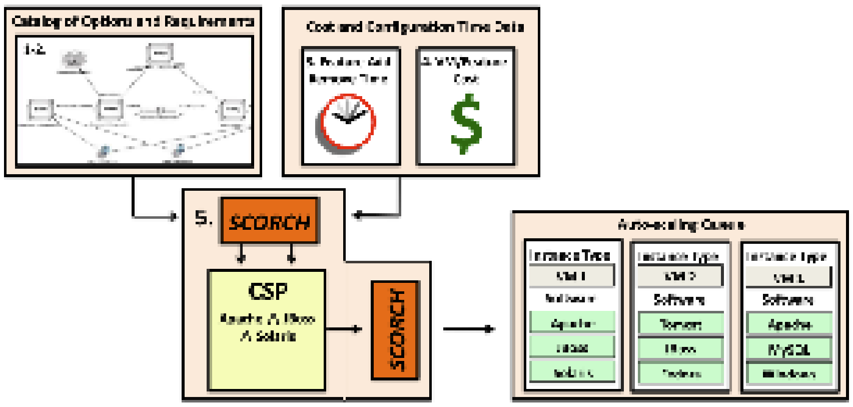


Figure 17: SCORCH Process

The SCORCH MDE process is shown in Figure 17 and is described below:

- Developers use a SCORCH *cloud configuration model* to construct a catalog of configuration options that are available to virtual machine instances.

- Each application considered in the auto-scaling queue configuration optimization provides a *configuration demand model* that specifies the configuration for each type of virtual machine instance the application will request during its execution lifecycle.

- Developers provide a *configuration adaptation time model* that specifies the time required to add/remove a feature from a configuration.

- Developers provide a *cost model* that specifies the cost to run a virtual machine configuration with each feature present in the SCORCH cloud configuration model.

The cloud configuration model, configuration demand models, and load estimation model are transformed into a CSP and a constraint solver is used to derive the optimal auto-scaling queue setup.

9.4 **Results Applying SCORCH**

This section presents a comparison of SCORCH with two other approaches for provisioning virtual machines to ensure that load fluctuations can be met without degradation of quality of service. We compare the energy efficiency and cost effectiveness of each of the approaches for provisioning an infrastructure for supporting a set of e-commerce applications. We selected e-commerce applications due to the high fluctuations in workload that occur due to the varying seasonal shopping habits of users. To compare the energy efficiency and cost effectiveness of these approaches, we chose the pricing model and available virtual machine instance types associated with Amazon EC2.

We investigated three-tiered e-commerce applications consisting of web front end, middleware, and database layers. The applications required 10 different distinct virtual machine configurations. For example, one virtual machine required JBOSS, MySql, and IIS/Asp.Net while another required Tomcat, HSQL, and Apache HTTP. These applications also utilize a variety of computing instance types from EC2, such as high memory, high-CPU, and standard instances.

To model the traffic fluctuations of e-commerce sites accurately we extracted traffic information from Alexa (`www.alexa.com`) for newegg.com (`newegg.com`), which is an extremely popular online retailer. Traffic data for this retailer showed a spike of three times the normal traffic during the November-December holiday season. During this period of high load, the site required 54 virtual machine instances. Using the pricing model provided by Amazon EC2, each server requires 515W of power and costs \$1.44 an hour to support the heightened demand (`aws.amazon.com/economics`).

9.4.1 Experiment: Virtual Machine Provisioning Techniques

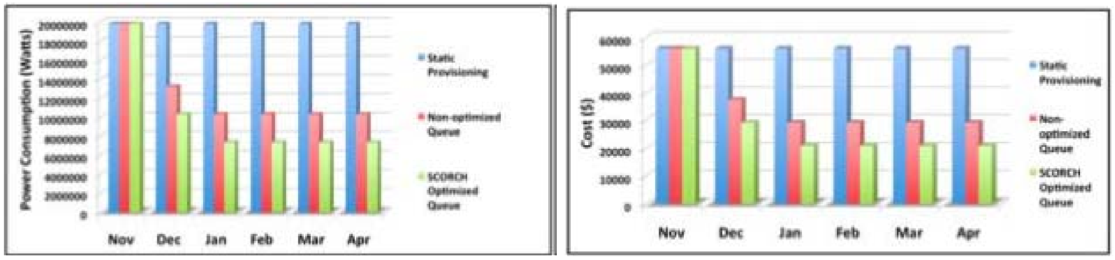
Static provisioning. The first approach consists of provisioning a computing infrastructure equipped to handle worst case demand at all times. In our scenario, this technique would require that all 54 servers were run continuously to ensure that response time is maintained. This technique is similar to computing environments that do not permit any type of auto-scaling. Since the infrastructure can always support the worst-case load, we refer to this technique as *static provisioning*.

Non-optimized auto-scaling queue. Another approach is to augment the auto-scaling capabilities of a cloud computing environment with an auto-scaling queue. In this approach, auto-scaling is used to adapt the number of resources to meet the current load that the application is experiencing. Since additional resources can be allocated as demand increases, we need not boot all 54 servers continuously. Instead, an auto-scaling queue with a virtual machine instance for each of the ten different configurations required by the application must be available to be allocated on demand. We refer to this technique as *non-optimized auto-scaling queue* since the auto-scaling queue is not optimized.

SCORCH. In this approach we use SCORCH to minimize the number of virtual machine instances required in the auto-scaling queue while ensuring that response time is met. By optimizing the auto-scaling queue with SCORCH, the size of the queue can be reduced by 80% to two virtual machine instances.

9.4.2 Power Consumption & Cost Comparison of Techniques

The maximum load for the 6 month period occurred in November and required 54 virtual machine instances to support the increased demand, decreasing to 26 servers in December and finally 18 servers for the final four months. The monthly energy consumption and operational costs of applying each response time minimization technique can be seen in Figure 18(a) and Figure 18(b), respectively. Since the maximum demand of the e-commerce applications required 54 virtual machine instances to function, the static provisioning technique consumed the most power and was the most expensive, with 54 virtual machine instances pre-booted at all times.



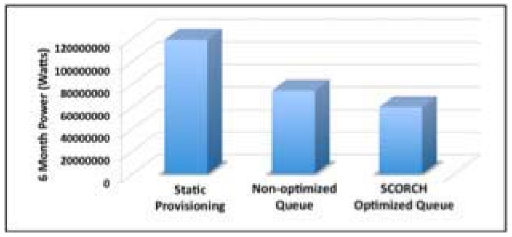
(a) Monthly Power Consumption (b) Monthly Cost

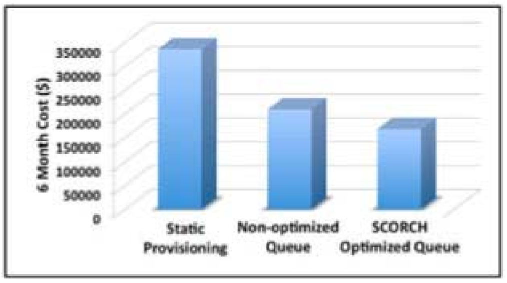
Figure 18: Monthly Power Consumption and Cost

The non-optimized auto-scaling queue only required ten pre-booted virtual machine instances and therefore reduced power consumption and cost. Applying SCORCH yielded the most energy efficient, lowest cost infrastructure by requiring only two virtual machine instances to be placed in the auto-scaling queue. Figure 19 compares the total power consumption and operating cost of applying each of the virtual machine provisioning techniques for a six month period. The non-

]

optimized auto-scaling queue and SCORCH techniques reduced the power requirements and price of utilizing an auto-scaling queue to maintain response time in comparison to the static provisioning technique.



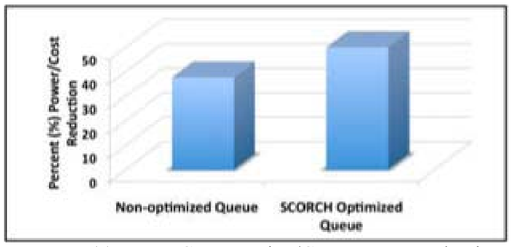


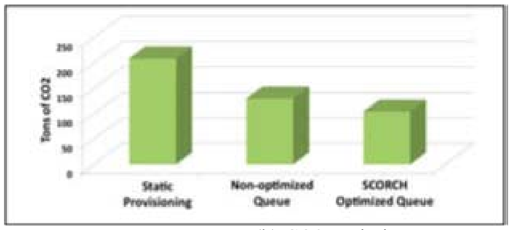
(a) Total Power Consumption (b) Total Cost

Figure 19: Savings in Power Consumption and Cost

The figure compares the savings of using a non-optimized auto-scaling queue versus an auto-scaling queue generated with SCORCH. While both techniques reduced cost by more than 35%, deriving an auto-scaling queue configuration with SCORCH yielded a 50% reduction of cost compared to utilizing the static provisioning technique. This result reduced costs by over $165,000 for supporting the e-commerce applications for 6 months.

More importantly than reducing cost, however, applying SCORCH also estimates reduction in CO2 emissions by 50% as shown in Figure 20(a). According to recent studies, a power plant using pulverized coal as its power source emits 1.753 pounds of CO2 per each kilowatt hour of power produced [57]. Therefore, as shown in Figure 20(b), not using an auto-scaling queue results in an emission of 208.5 tons of CO2 per year. Applying the SCORCH optimized auto-scaling queue, however, cuts emissions by 50% resulting in an emission reduction of 104.25 tons per year.





(a) Power Consumption/Cost Percent Reduction (b) CO2 Emissions

Figure 20: Environmental Impact of SCORCH-based Deployment

9.5 Concluding Remarks

Auto-scaling cloud computing environments help minimize response time during periods of increased application demand, while reducing cost during periods of light demand. The time to boot and configure additional virtual machine instances to support applications during periods of high demand, however, can negatively impact response time. This report describes how the *Smart Cloud Optimization of Resource Configuration Handling* (SCORCH) MDE tool uses feature models to represent the configuration requirements of multiple software applications and the power consumption/operational costs of utilizing different virtual machine configurations, transforms these representations into CSP problems and analyzes them to determine a set of virtual machine instances that maximizes auto-scaling queue hit rate. These virtual machine instances are then placed in an auto-scaling queue so that response time requirements are met while minimizing power consumption and operational cost.

The following are lessons learned from using SCORCH to construct auto-scaling queues:

- **Auto-scaling queue optimization effects power consumption and operating cost.** Using an optimized auto-scaling queue greatly reduces the total power consumption and operational cost compared to using a statically provisioned queue or non-optimized auto-scaling queue. SCORCH reduced power consumption and operating cost by 50% or better.

- **Dynamic pricing options should be investigated.** Cloud infrastructures may change the price of procuring virtual machine instances based on current overall cloud demand at a given moment. Our future work is incorporating a monitoring system to allow SCORCH to take advantage of such price drops when appropriate.

- **Predictive load analysis should be integrated.** The workload of a demand model can drastically effect the resource requirements of an application. In future work, SCORCH will take into account predictive load analysis to auto-scaling queues that cater to the workload characteristics of applications.

SCORCH is part of the ASCENT Design Studio and is available in open-source format from code.google.com/p/ascent-design-studio.

Future Work

For the near term future we are also investigating issues and challenges in mobile cloud environments. One important topic we are investigating pertains to maximizing the uptime of mobile devices that provide critical services in a Cloud. We expect to present data in this regard in our subsequent report.

10. Deployment Optimizations for Mobile DoD Systems

This section describes our effort which continues the theme of auto-scaling issues in cloud computing, however, with a focus on investigations on battery power concerns for smart phone-based cloud computing environments. Such research is important for the US Air Force, which is increasingly adopting the use of new technologies including smart phones for national security.

10.1 Mobile Devices in DoD Systems

Current trends and challenges: The unprecedented growth in smart phone technology is giving rise to new applications that illustrate non-conventional usage of smart phones [65]. For example, these applications may include situational awareness in military-centric operations (e.g., the DARPA Transformative Apps program), emergency services, disaster search-and-recovery, and intelligent transportation. Consider, for example, the natural disasters of 2010 like the Haiti earthquake or the massive flooding in the state of Tennessee. In both these situations, most of the infrastructure, such as the roads and phone services (both landline and cellular), and utilities, such as gas and electricity, were rendered unavailable. A number of instances of smart phone usage for survival have come to light in the days following the calamity.

It is conceivable, therefore, to think of forming ad hoc networks of smart phones carried by search-and-rescue teams as the best means in these circumstances to identify survivors trapped under the debris or those trapped in their houses due to raging flood waters, and coordinate the rescue operations. To operationalize smart phone-based search-and-rescue missions, it is necessary for the collection of smart phones – *a smart phone cloud* -- involved in the mission to be able to support a group of real-time services that provide distributed sensing operations, data correlation capabilities stemming from acquisition of distributed streams of images, audio and video, and location-based services.

However, since these smart phones have limited battery life and hardware resources, keeping the collective set of services that make up the mission capabilities up and running for the maximum amount of time is crucial for maximizing the chances of finding more survivors. Maximizing the mission lifespan is important because the smart phones operated by first responders are often deployed in environments where readily replenishing the resources, such as batteries, is infeasible. Despite these constraints, key quality of service (QoS) requirements of real-time and reliable dissemination of information to the concerned stakeholders, such as first responders in search-and-rescue missions, must be met.

The requirements outlined above can be met by effectively deploying the services that make up the mission on the collection of smart phones involved in the mission. Here ad hoc network can be formed by smart phones which are self-configuring and self-organizing as no physical infrastructure is available for forming a centrally administered wireless network. Hence we are assuming that the appropriate ad hoc routing protocols like AODV [66], DYMO [67], etc. are available for routing of data to/from ad hoc network. Also, each node in the ad hoc network has equal probability of acting as hosts as well as routers to route data to/from other nodes in the network. The reason is that considerable battery power is consumed in routing data to/from the network to/from the outside network. Thus if a particular node has a higher probability of acting as a router, then its battery power will be drained faster, which can render the entire distributed application nonoperational earlier than its maximized service uptime.

Such a deployment problem is hard for two reasons. First, assuring the timely and reliable dissemination of information in operating environments where availability of resources, such as networks, is unpredictable requires deploying the individual services on the collection of smart phones in a way that will ensure the schedulability of the services while efficiently using the scarce resources. Secondly, the rate of drain of battery charge adds a new dimension of challenges to an already challenging problem because battery drain is often dictated by the amount of computation and communication activities.

Solution approach → Service Uptime Maximization in Smart phone Clouds. In this research we focus on solving the service uptime maximization problem, which is the problem of ensuring that the operational capability of the mission provided by the collection of services deployed on the group of smart phones remains up and running for the maximum duration of time. In other words, it is necessary to minimize the rate at which the smart phone batteries drain themselves. Since every service (and its software components) of the mission consumes different computational and communication resources of the smart phone, battery drain is impacted differently. Hence, the service uptime maximization problem requires solving the deployment problem that minimizes battery drain (or preserves the battery charge) while also satisfying the QoS requirements.

To address these challenges, we present a deployment framework called SmartDeploy, which extends the earlier work on ScatterD [47]. ScatterD combined binpacking heuristics with evolutionary algorithms [68] to minimize power consumption in nodes. It overcame the limitations of applying each of these algorithms in isolation. In particular, ScatterD provided a first-fit heuristic bin packer which places each item into the first available bin in which it will fit. In the case of maximizing service uptime, the software components of the services must be deployed in a way that minimizes battery drain on each smart phone. A first-fit heuristic may not necessarily find the right solution to our problem.

Consequently, SmartDeploy provides a framework that can be strategized with the desired bin packing heuristic along with a strategizable framework to plug in the desired evolutionary algorithm so that a variant of the hybrid algorithm can be synthesized. To solve the service uptime maximization problem, SmartDeploy is strategized with the worst-fit bin packer which ensures that services are load balanced across the collection of smart phones used in the mission in a way that minimizes battery drain while also delivering the required QoS. The evolutionary algorithm generates initial random vectors and evaluates them using a fitness function. In this research we have limited ourselves to offline deployment of services assuming that the rescue missions and their parameters are planned *a priori*. The case of determining an effective deployment at runtime is orthogonal to the focus of this research and is the focus of future work, which will require additional runtime protocols involving message exchanges among participating smart phones. We believe that the polynomial runtime complexity of SmartDeploy can make it a promising approach even at runtime.

10.2 Challenges in Maximizing Smart phone Cloud Lifetime

In this section we use an example of a video recognition service for disaster monitoring as a case study to highlight the challenges in maximizing the service uptime for smart phone-based distributed, real-time systems. Figure 21 shows an example of a distributed video recognition service used in disaster monitoring and recovery. The service comprises of different software components like video capturing (C1), segmentation (C2), feature extraction (C3), tracking (C4), activity analysis (C5) and information dissemination (C6). Each of these software component has different hardware resource requirements, such as memory and CPU, and different power consumption rates. For simplicity, we have shown one such distributed service (video recognition) consisting of six software components and four smart phones for disaster monitoring. Out of four smart phones used, two of them are Android-based HTC phones and the other two are iPhones. Software components C1, C4, and C5 can be executed only on Android-based smart phones, while software components C2, C3, and C6 can be executed only on iPhones. In general, a disaster monitoring service can be composed of a combination of services such as distributed image recognition and distributed location-based services. Such a comprehensive service can consist of hundreds of software components deployed onto hundreds of smart phones. The deployment plan, which comprises a mapping of the software components of the services to the smart phones, should meet both the hardware resources constraints and power constraints such that the service can last for as much time as possible while also meeting the real-time application requirements.

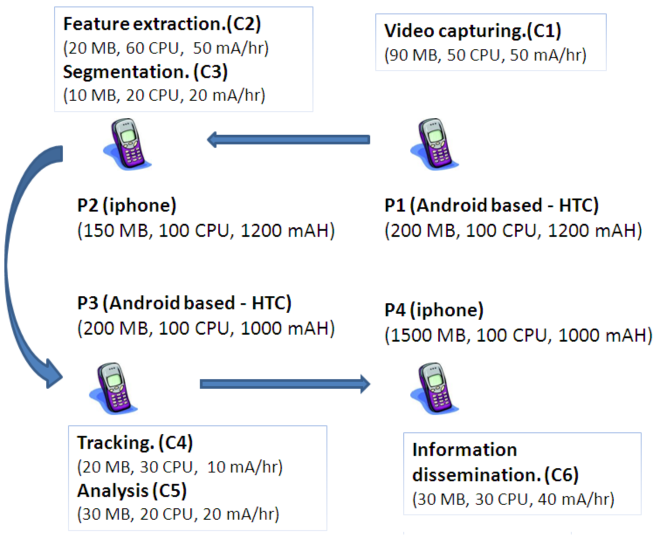

Feature extraction.(C2)
(20 MB, 60 CPU, 50 mA/hr)
Segmentation. (C3)
(10 MB, 20 CPU, 20 mA/hr)
Video capturing.(C1)
(90 MB, 50 CPU, 50 mA/hr)
P2 (iphone)
(150 MB, 100 CPU, 1200 mAH)
P1 (Android based - HTC)
(200 MB, 100 CPU, 1200 mAH)
P3 (Android based - HTC)
(200 MB, 100 CPU, 1000 mAH)
P4 (iphone)
(1500 MB, 100 CPU, 1000 mAH)
Tracking. (C4)
(20 MB, 30 CPU, 10 mA/hr)
Analysis (C5)
(30 MB, 20 CPU, 20 mA/hr)
Information
dissemination. (C6)
(30 MB, 30 CPU, 40 mA/hr)

Figure 21: Smart phone Cloud Motivating Scenario

Challenge 1: Dealing with complex hardware/software design constraints. In our case study example of the distributed video recognition service, its software components have different hardware and software resource requirements. For example, the video capturing component requires high memory and communicational power as it stores the captured video and sends it to the phone hosting feature extraction and segmentation components. The feature extraction and segmentation components require high CPU and computational power as they run complex algorithms based on extraction and segmentation on the video. The tracking and activity analysis components are involved in significant communication activities that consume battery power as they constantly communicate with the phone hosting the information dissemination component. A disaster monitoring system comprises many distributed applications consisting of hundreds of smart phones and hundreds of software components hosted on them. How these software components are deployed on the devices will determine how long the overall mission will last, because the uptime of the mission depends on how long the batteries last.

In general, network embedded devices like smart phones have limited battery power and limited hardware resources like CPU and memory. Moreover, the software components deployed on these devices consume power at different rates, which is governed by the computation and communication activities induced by the software components. The amount of time a software component runs is directly proportional to the amount of battery power available to it with sufficient hardware resources. Thus, the power consumption rate of these software components, and what devices they get deployed on are the key factors that affects the service uptime. Given that a mission is realized by distributing its services across a group of smart-phones, keeping the entire distributed application up for a longer duration is challenging because even if one of the smart phone's battery is exhausted, then the software components deployed on it are no longer available which makes the overall distributed system no longer work.

Thus, a deployment plan should be generated such that each of the software components gets maximum available power and sufficient hardware resources which will maximize the overall service uptime of the mission. In generating such a deployment plan, we must consider both the computational and communication power consumption rates of the software components.

Challenge 2: Dealing with heterogeneity of available resources and execution constraints. Our case study example illustrates heterogeneity in the smart phone hardware and operating systems. It is conceivable that embedded devices such as smart phones used in mission-critical applications such as disaster search and rescue management have different available hardware resources like CPU type, available memory, and lifetime of battery. Due to this heterogeneity, certain software can execute on only certain devices. For example, *smart phone apps* developed for *iPhones* cannot execute on Android-based phones. As outlined in Challenge 1, the deployment topology of these mission-critical systems must address various design constraints like power capacity, memory, and CPU, which is a hard problem. The problem becomes even harder with the heterogeneity of the platforms and the software execution constraints. In the case study example, there are two Android-based HTC phones and the other two are iphones. Moreover, independent software components for video capturing, tracking and analysis can execute only on Android-based phones. Similarly, feature extraction, segmentation and information dissemination can execute only on iPhones. Such constraints affect the deployment plan which in turn affects maximizing service uptime.

Challenge 3: Dealing with scale of the system. The case study example of distributed video recognition service is comprised of four devices hosting six software components which means there exist 6^4 possible deployment plans (if there are no restrictions). Several optimization techniques are available to solve the deployment challenges explored in Challenges 1 and 2 described above. The solutions can be characterized and solved using constraint satisfaction programming (CSPs) [69], integer programming [70] and Bender's decomposition [71]. Although our case study represents a very small problem size which can be solved by bin-packing heuristics, integer programming or evolutionary algorithms, typical mission critical applications will comprise several hundreds of devices and many more software components. Thus, when the problem size scales to 300^{100} or even more, and moreover, considering additional hardware and software design constraints, as outlined in Challenges 1 and 2, many of the known techniques cannot readily scale to hundreds of software components and hundreds of devices. In other words, the solutions are computationally very expensive to obtain.

Bin packing heuristics have been developed to overcome these challenges to produce valid deployment plans; however, these plans do not necessarily produce the optimal solutions for large problem sizes. Evolutionary algorithms are commonly used in deployment optimization problems. However their performance degrades when the solution space is huge and has tight constraints that lead to a large number of invalid points in the search space.

10.3 The SmartDeploy Architecture

To address the challenges described in Section 10.2, we use a hybrid algorithm that integrates bin packing heuristics with evolutionary algorithms (e.g., particle swarm optimization PSO [40] and genetic algorithms [72]) so that we can reap the benefits of both while overcoming the limitations of individual techniques. Moreover, rather than fixing a specific heuristic or an evolutionary algorithm, we provide a framework that enables a deployment planner to strategize the framework with the desired techniques. The advantage of using bin-packing heuristics is that they produce a valid deployment topology while the advantage of using evolutionary algorithms is that they explore multiple solutions in the design space.

Here we show the architecture of SmartDeploy, which is a strategizable framework for deployment planning that addresses the three challenges described in Section 10.2. The SmartDeploy framework is applied to solve the Service Uptime Maximization problem. Figure 22 shows the SmartDeploy framework combining the worst-bin packer and PSO algorithm. It shows a generic interface to encode objective functions and constraints, and the hybrid algorithm to solve design-time constraint optimization problems. The algorithm for combining the worst-fit bin packer and genetic algorithm is also similar. The white colored blocks show the newly added features by SmartDeploy, blue colored blocks show the integration between original and new features and the grey colored blocks show the original features of ScatterD.

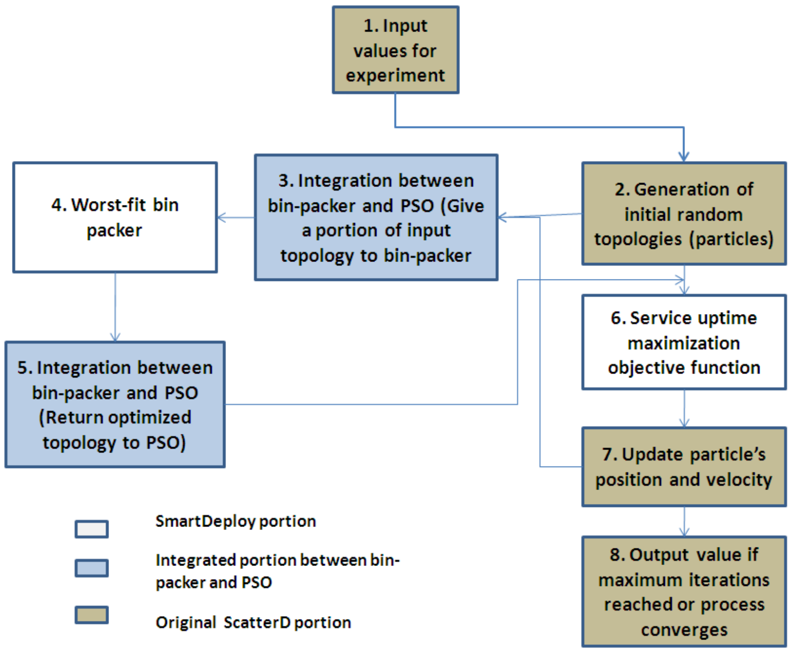

1. Input values for experiment
2. Generation of initial random topologies (particles)
3. Integration between bin-packer and PSO (Give a portion of input topology to bin-packer
4. Worst-fit bin packer
5. Integration between bin-packer and PSO (Return optimized topology to PSO)
6. Service uptime maximization objective function
7. Update particle's position and velocity
8. Output value if maximum iterations reached or process converges
SmartDeploy portion
Integrated portion between bin-packer and PSO
Original ScatterD portion

Figure 22: SmartDeploy Architecture

10.4 Empirical Results using SmartDeploy

We compared the deployments produced by five different deployment techniques. The five techniques we compared are:

- *Worst-fit bin packing* -A worst-fit heuristic of bin-packing algorithm.
- *Particle Swarm Optimization (PSO)* - Only PSO algorithm from SmartDeploy framework.
- *SmartDeploy PSO* -The PSO variant of SmartDeploy which combines worst-fit bin-packer with PSO algorithm.
- *Genetic* - Only Genetic algorithm algorithm from SmartDeploy framework.
- *SmartDeploy Genetic* -The genetic variant of SmartDeploy which combines worst-fit bin-packer with genetic algorithm.

The experiments were conducted on a single Windows XP desktop with 2.19 GHz Intel Core 2 Duo processor and 2 GB RAM. Java Virtual Machine (JVM) version 1.6 was used for the experiments. For both PSO and genetic algorithm, a population size of 20, local learning coefficient of 0.5, global learning coefficient of 2, and 20 search iterations (generations) were used. The genetic algorithm allowed a total of 10% of the population to be passed through to the next generation, selected the top 25% of solutions for mating, and applied a mutation probability of 5%. A uniform distribution for generating initial random vectors is used to cover more area and not inadvertently bias our search to a specific region. Experiments 1 and 2 described below were conducted using 100 nodes and 100 software components. The number of nodes tested for the experiment ranges from 30 to 100. The number of software components is kept constant.

10.4.1 Experiment 1: Homogeneous nodes, heterogeneous software components

The first experiment was conducted using homogeneous nodes. That is, each node has the same amount of memory and power capacity. The software components deployed on the nodes were heterogeneous (each component required a different amount of memory and power consumption capacity). Here, constraints were placed on the amount of memory available on all nodes and the amount of memory required by all software components, such that total hardware and software resource requirements should not exceed total availability.

We hypothesize that SmartDeploy should provide a significant increase in service uptime compared to the bin-packing algorithm and PSO. Here, although the nodes have homogeneous properties for the amount of memory and the battery power capacity, the heterogeneous properties of the software components *i.e.*, each of them requiring different amount of memory and power consumption capacity causes SmartDeploy to produce better results than the worst-fit bin packer and evolutionary algorithms alone.

Service uptime is plotted against the number of homogeneous nodes in Figure 23. The SmartDeploy algorithms provide 94% and 58% improvement in maximizing service uptime over PSO and genetic algorithms, respectively. However, it gives only 20% improvement over worst-fit bin packer. Due to the homogeneous properties of the nodes, the worst-first bin packer gives better results as compared to both of the evolutionary algorithms, and are close to that of SmartDeploy.

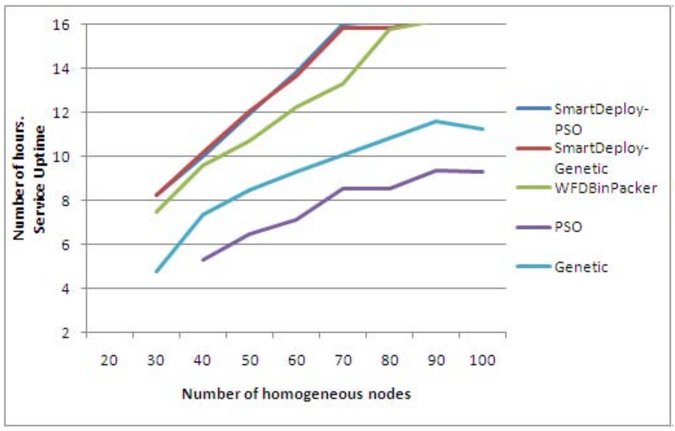


Figure 23: Homogeneous Nodes, Heterogeneous Components

10.4.2 Experiment 2: Heterogeneous nodes (but same OS), heterogeneous software components

The second experiment was conducted using heterogeneous nodes. That is, half the number of nodes have one set of properties while the other half has another set of similar properties. For lack of space we do not report on other variations. The software components deployed on the heterogeneous nodes were also heterogeneous - each component required a different amount of memory and power consumption capacity. Constraints were placed on the amount of memory available on all nodes and the amount of memory required by all the software components, such that total hardware and software resource requirements would not exceed total availability.

We surmise that SmartDeploy should provide significant improvement in service uptime compared to the bin-packing algorithm and evolutionary algorithms. Nodes having heterogeneous properties for the amount of memory and battery power capacity, and having heterogeneous properties for the software components (each component requiring a different amount of memory and power consumption capacity), should cause the SmartDeploy algorithms to produce better results than the worst-fit bin packer and evolutionary algorithms alone.

As seen in Figure 24, due to the heterogeneous properties of nodes and software components, and large problem size, the performance of evolutionary algorithms degrades. PSO gives invalid topologies in this scenario. Genetic algorithm gives invalid topologies when software components are tightly packed onto devices. Even when the number of devices increases, SmartDeploy algorithms provide up to 162% better service uptime. They also provide up to 75% more service uptime than worst-fit bin packer.

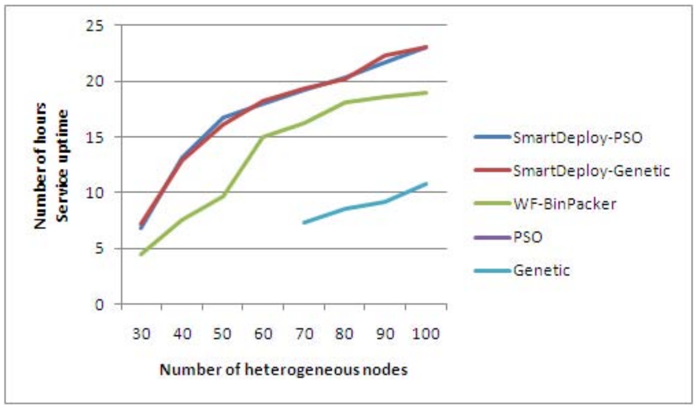


Figure 24: Heterogeneous Nodes, Heterogeneous Components

10.4.3 Experiment 3: Varying the number of software components (heterogeneous) deployed on fixed number of heterogeneous nodes

The third experiment was conducted by varying the number of heterogeneous software components being deployed on fixed number of heterogeneous nodes. The number of software components varied from 100 to 200 with increments of 20. Constraints were placed on the amount of memory available on all nodes and the amount of memory required by all the software components, such that the total hardware and software resource requirements do not exceed their total availability.

Our hypothesis is that as the number of software components increases, the topologies become tightly constrained. If the solution space increases, then it should cause the bin-packer to provide a less than optimal value. The tightly constrained solution space should cause evolutionary algorithms to degrade in their performance. As seen in Figure 25 the devices become tightly packed with increasing number of software components and constraint on memory requirements. The evolutionary algorithms degrade in performance and give invalid deployment topologies. The SmartDeploy algorithms give up to 50% more service uptime as compared to worst-fit bin packer.

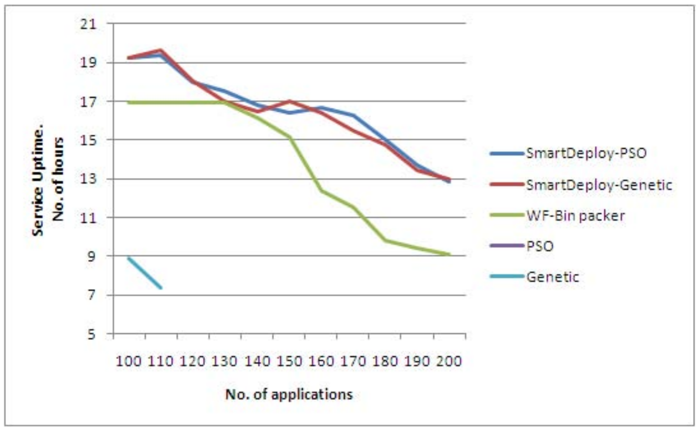


Figure 25: Varying the Number of Heterogeneous Software Components

10.4.4 Experiment 4: Heterogeneous nodes (including different OS) and heterogeneous software components

The fourth experiment was conducted using heterogeneous nodes, such that 30% of nodes had one set of properties while 70% of nodes had another set of similar properties. Also, a different OS (Android-based and iPhone) was used for each set. The software components deployed on them were also heterogeneous. Each component required different memory, power consumption capacity, and execution platform (OS). Constraints were placed on the execution platform (OS), amount of memory available on all nodes, and the amount of memory required by all software components, such that total hardware and software resource requirements would not exceed total availability.

Our hypothesis is that SmartDeploy should provide significant improvement in service uptime compared to the bin-packing algorithm and evolutionary algorithms. Here the nodes having heterogeneous properties for the amount of memory, the battery power capacity and execution platform (OS), the heterogeneous properties for the software components, *i.e.*, each of them requiring different amount of memory and power consumption capacity and execution platform (OS) should cause the SmartDeploy algorithms to produce better results than the worst-fit bin packer and evolutionary algorithms alone.

Figure 26 displays service uptime as a function of the number of heterogeneous nodes. The performance of evolutionary algorithms degrades due to the heterogeneous properties of nodes and software components, and large problem size. PSO gives invalid topologies in this scenario. A genetic algorithm gives invalid topologies when software components are tightly packed onto devices. SmartDeploy algorithms give higher service uptime than bin-packer and the evolutionary algorithms.

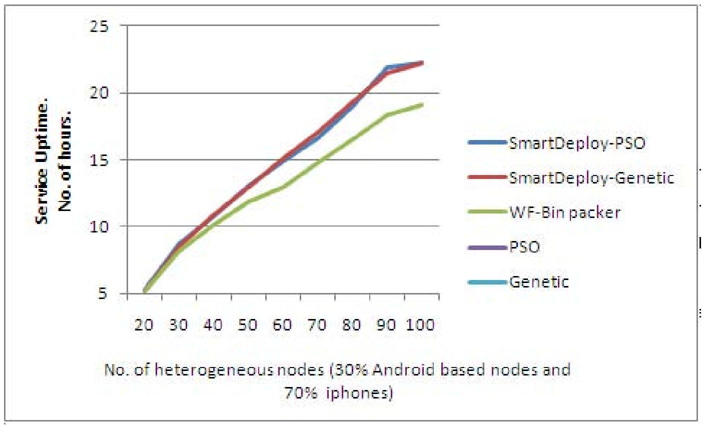


Figure 26: Heterogeneous Nodes (Different OS), Heterogeneous Components

10.4.5 Experiment 5: Comparison of service uptime by all the algorithms with that of brute-force algorithm

We attempted to obtain the optimum service uptime using a brute-force algorithm which tries each and every combination of deployment topologies. However, we observed that running the brute-force algorithm even for even small problem sizes takes significant time. So it was not practical to run it for large problem sizes of hundreds of nodes and hundreds of software components. Table 1 shows the running time for the brute-force algorithm over a small problem size.

Nodes	Software components	Service uptime(msec)
5	5	78
5	7	1219(1.2 secs)
5	9	33312(33.3 secs)
5	11	1261211(21 minutes)

Table 1: Time taken to run Brute-force algorithm for service uptime

10.4.6 Experiment 6: Comparison of computation time taken by each of five algorithms to execute

The sixth experiment was conducted to observe the average time taken by each of the five algorithms to execute. Here the experimental values used in Experiment 2 were used regarding heterogeneous nodes and heterogeneous software components. The average values for service uptime for the entire range of nodes were recorded. As seen in Figure 27, worst-fit bin packer takes least amount of time to run (47 milliseconds). The SmartDeploy algorithms take the most amount of time to run, between 2,000 milliseconds to 3,200 milliseconds. Since we are considering an offline solution for deployment topology, a delay of a few seconds is tolerable to achieve better service uptime. Hence the use of SmartDeploy algorithms is desirable in such situations.

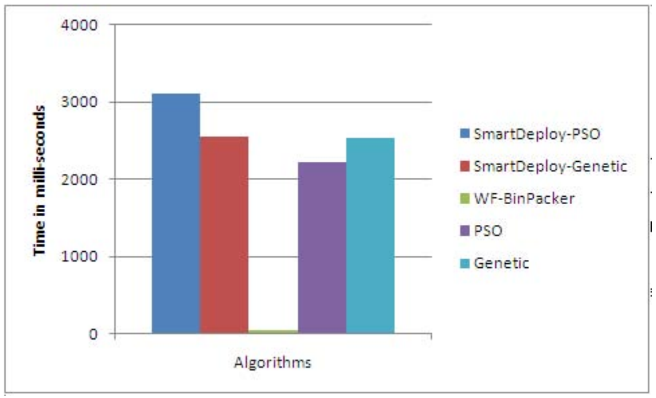


Figure 27: Time Taken by Each Algorithm to Execute

10.5 Concluding Remarks

Service uptime maximization in distributed applications hosted on a network of smart-phones can be achieved through effective deployment. Several optimization techniques are commonly used for deployment problems in distributed real-time and embedded systems. Algorithms with exponential runtime complexity like integer programming are not scalable when the problem size increases up to hundreds of devices. Bin-packing heuristics tend to generate valid deployment topologies, but they may not give optimal solutions when problem size increases. Evolutionary algorithms are commonly used for deployment problems since they explore a variety of design solutions. However, as the number of constraints and the problem size increases, they tend to degrade in performance and returned invalid results.

This section describes a framework called SmartDeploy that provides a hybrid deployment technique to achieve service uptime maximization. It builds upon the earlier work, called ScatterD, which combines first-fit bin packer with the evolutionary algorithm to reduce power consumption in DRE systems. SmartDeploy enables a user to strategize both the evolutionary algorithm as well as the bin packing heuristic. A concrete manifestation of SmartDeploy using the worst-case bin packer along with evolutionary algorithms is presented to solve the service uptime maximization problem for smart phone-based mission critical applications.

Using worst-fit bin packer heuristic, the software components of the distributed application can be evenly deployed on the available devices such that they can obtain maximum available battery power and sufficient hardware resources. The experimental results show that SmartDeploy framework increased service uptime from 20% to 162% beyond that provided by worst-fit bin packer and evolutionary algorithms when used independently. The following lessons were learned conducting this research:

Since the running time of the SmartDeploy algorithms is only slightly more than the algorithms we compared against, it is practical to use the hybrid algorithm. In future work we intend to investigate the use of the SmartDeploy framework in runtime deployment decisions. We also intend to investigate other distribution techniques for the generation of initial random topologies of evolutionary algorithms like Gaussian distribution to see if they can achieve better solutions.

We intended to run the brute-force optimal algorithm to compare the service uptime solutions from each of the five algorithms we used in our experiments to see how our solutions compare to the optimal one. However, we observed that running the brute-force algorithm, even for small problem sizes, takes considerable amount of time. Hence it was not practical to test it out for the large problem size that we use.

11. Dynamic Resource Management for Cloud Computing-based DoD Systems

Our effort continued the theme on the auto-scaling issues in cloud computing with a focus on investigations into balancing the costs of acquiring and releasing machines, and their reconfigurations. An additional dimension of our research focused on deployment and configuration of distributed systems, which can also handle deployment in the cloud – which can be a tactical cloud, used in US Air Force operations.

11.1 Introduction

Large enterprise software systems such as eBay, Priceline, Amazon and Facebook need to provide high assurance in terms of Quality of Service (QoS) metrics such as response times, high throughput, and service availability to their users. Without such assurances, service providers of these applications stand to lose their user base, and hence their revenues. Typically customers maintain Service Level Agreements (SLAs) with service providers for the QoS properties. Failure to comply with satisfying these QoS metrics leads to a major loss of revenue in the form of decreased user base [73]. Similar requirements exist in the context of DoD applications, where the QoS requirements become even more stringent, and instead of revenues the DoD is concerned with cost-effective solutions. In the rest of the section we will use enterprise applications to drive home our point and the defined solutions.

Catering to the SLA while still keeping costs low is challenging for such enterprise systems due primarily to the varying number of incoming customers to the system. For example, consider Figure 28 which depicts a real-world scenario wherein workload of the FIFA 1998 soccer world cup website in the number of incoming clients to such a website is highly varying depending upon a number of factors such as time of day, day of week and other seasonal factors.

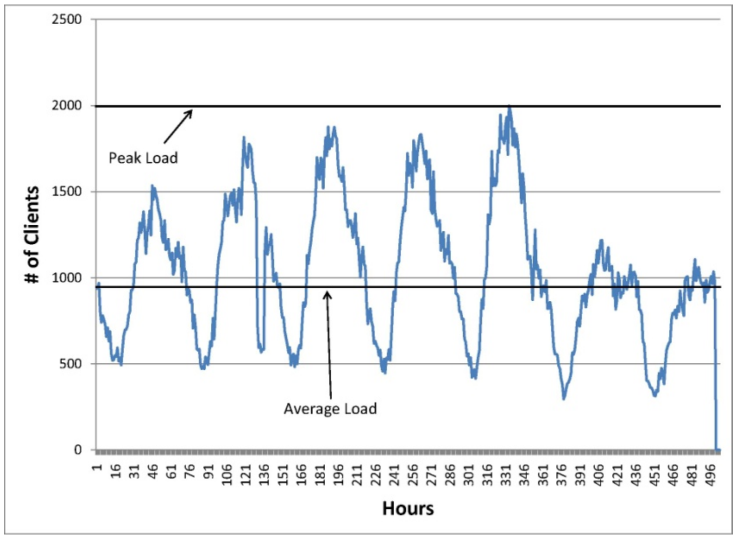


Figure 28: Workload for the 1998 FIFA Soccer World Cup

Such a workload is very typical of all commercial websites, and planning capacity for such workload is not easy. Capacity could be planned for the average load or for peak load. When planned for the average load, there is less cost incurred due to less hardware used but performance will be a problem when peak load occurs. Bad performance will discourage customers and revenue will be affected. On the other hand if capacity is planned for peak workload, resources will remain idle most of the time.

Auto-scaling supported in cloud computing environments overcomes these challenges. Cloud computing providers such as Amazon EC2 provide access to hardware which can be allocated or deallocated at any time. Images of client software can be created beforehand which can be loaded onto the machine. When not required, the same machines can be released. Machine usage costs on an hourly basis. Amazon EC2 provides an API which can be used to automate this process. A problem with such a resource allocation scheme is the chance of thrashing where due to frequent variation of workload, machines can be added and released on every sample.

A desirable solution would require an ability to predict the incoming workload on the system and allocate resources *a priori*. This capability in turn will enable the application to be ready to handle the load increase when it actually occurs. A corollary requirement is the need to identify how many machines should actually be provisioned and started to handle the predicted load. For example, consider a situation where there is N number of machines already running and handling M customers for a given application. Suddenly, the number of customers increases to M + 100 and processor utilization also increases in the running nodes. Naturally, this situation requires increasing the number of machines allocated, but by how much is unknown. Anything less will provide degraded performance; anything more implies cost incurred by the customer for resources not actually used by the application.

In summary, auto-scaling the resources in a cloud environment is not an easy and straightforward task. Overcoming these challenges will require algorithms which take into account the following: (i) overheads related to state transition when number of resources are changed, (ii) ability to accurately predict future workload, and (iii) compute the right number of resources required for the expected increase or decrease in workload. This report describes a resource allocation algorithm based on model predictive techniques which allocates or deallocates machines to the application based upon optimizing the utility of the application over a limited prediction horizon.

11.2 Challenges to Elastic Resource Provisioning in Cloud Environments

This section discusses the challenges to realizing elastic resource provisioning in large-scale component-based systems. Many of the challenges that are faced in elastic resource provisioning using auto-scaling can be highlighted from the workload pattern in Figure 28. In DoD scenarios, due to the fluctuating availability of resources, it becomes important to support an elastic resource provisioning capability so that mission-critical applications can continue to execute in a cost effective manner.

11.2.1 Challenge 1: Workload Forecasting

The auto-scaling strategy in a cloud environment will involve acquiring and release of resources as the workload imposed by the application changes with time. Releasing resources is easy, however, acquiring resources incurs performance overheads due to the following reasons. First, there is a need to make a call on the cloud API which starts the acquisition process. The machines will then need to boot up with the specified image, the application(s) needs to be started, and there also might be the need for state update. Thus, it is desirable if the resources can be acquired earlier than the time when workload actually increases. This outcome can be possible only if the future workload can be predicted, possibly using historical data.

11.2.2 Challenge 2: Identify Resource Requirement for Incoming Load

Figure 28 plots the number of customers who use the system every hour. Since the number of customers varies every hour, the number of resources required also varies. The required number of resources is a function of the number of customers, the nature of the application, and also the type of calls that each customer makes on the application. The resources required need to be estimated properly so that they can be provisioned within the cloud infrastructure. The resource estimation also needs to be very accurate. If it is not accurate then there is the potential of under- or over-provisioning of resources, each of which has its pitfalls.

11.2.3 Challenge 3: Resource Allocation while Optimizing Multiple Cost Factors

To optimize resource usage and/or minimize idle resources, an ideal solution is to define a time interval and change resources as many times as possible as workload changes. In the limit this interval could be made infinitesimally small and resources are changed continuously in accordance with the change in load, assuming we can always at least over estimate the load. This extreme will obviously ensure that the optimum number of resources is always used. Obviously, such a scheme is not possible since changing resources is not spontaneous. Challenge 1 highlights the overhead in allocating a resource. Thus, scaling resources up or down also involves cost and needs to be optimized.

11.3 Auto-Scaling Resources Using Look-Ahead Optimizations

Control theory offers a promising methodology to address the challenges described in Section 11.2. It allows systematically solving a general class of dynamic resource provisioning problems using the same basic control concepts, and to verify the feasibility of a control scheme before deployment on the actual system. In more complex control problems a pre-specified plan called the feedback map becomes inflexible and does not adapt well to constantly changing operating conditions. Therefore, researchers have studied the use of more advanced state-space methods adapted from model predictive control [74] and limited look-ahead supervisory control [75] to manage such applications [76–78]. These methods offer a natural framework to accommodate the above described system characteristics, and take into account multi-objective non-linear cost functions, finite control input sets and dynamic operating constraints while optimizing application performance. The autonomic approach proposed in [78], [79] describes a hierarchical control based framework to manage the high level goals for a distributed computing system by continuous observation of the underlying system performance. The key differences between these previous works and our work are in the nature of the performance models.

The auto-scaling algorithm presented in this report does not use a reactive strategy. Instead it provides a predictive solution leveraging concepts proposed by Sherif, et. al.[80], [81], which is applicable to systems that exhibit a hybrid behavior comprising both discrete-event and continuous dynamics and have a possibly large but finite set of control options. Formally, it has been shown before in the literature that dynamics of such systems can be captured using the model of switching hybrid systems. It is known that for such systems a multi objective control problem can be solved by using a limited look-ahead controller algorithm [80], [81], which is a type of model predictive control. This is done by selecting actions that optimize system behavior over a limited prediction horizon. The rest of this section describes our approach and shows how we resolve the three challenges described in Section 11.2.

11.3.1 Workload Prediction

To apply model predictive control ideas to the problem discussed, we predict the workload on the application and estimate the system behavior over the prediction horizon using a performance model. The optimization of the system behavior is carried on by minimizing the cost incurred to the application. This cost is a combination of various factors such as cost of SLA violations, leasing cost of resources and a cost associated with the changes to the configuration. The advantage of such a method is that it can be applied to various performance management problems from systems with simple linear dynamics to complex ones. The performance model can also be varied and corrected with system dynamics as conditions in the environment like workload variation or faults in the system change.

In our strategy, workload prediction is needed to estimate the incoming workload of the system for future time periods. Thankfully a number of techniques already exist in literature that can be applied for forecasting the traffic incident on a service. We used a second order autoregressive moving average method (ARMA) filter for the workload shown in Figure 28. The equation for the filter used is given by

$$\lambda(t+1) = \beta \times \lambda(t) + \gamma \times \lambda(t-1) + (1-(\beta+\gamma))(\lambda(t-2)) \tag{1}$$

The value for the variables β and γ are given by the values 0.8 and 0.15, respectively. Figure 29 shows the predicted workload compared to the actual workload.

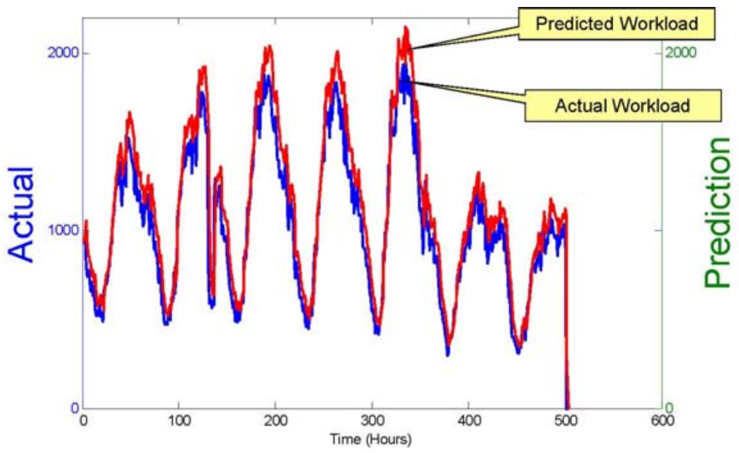


Figure 29: Predicted versus Actual Workload

11.3.2 Performance Model

The next challenge we resolve is identifying resource requirements for the predicted workload. The right number of resources will provide the desired response times (or other performance metrics) of the applications. For the above look-ahead framework we focus on response times of the application under different hardware configurations. The workload used in this work is the number of users currently in the system. It also depends upon what each user executes on the system. For example, some users could be browsing while some users could be entering data in a form. In our prior work [82] we have used Customer Behavior Modeling Graphs (CBMG) to model the overall behavior of customers. A CBMG is built from a log of previous customer behavior and computes the probability of a typical user to visit each page. Using this information, we can calculate the number of visits to a single page from the total number of customers in the system. The number of visits to each page helps in calculating the average load on each page.

Our prior work also developed analytical models to accurately estimate response times, which are used in Algorithm 1. This algorithm accepts the amount of workload given by a vector of client populations, each member representing the number of clients in each job class. The number of machines providing the service demand of the components and the think time for clients is also given as input. Algorithm 1 initially creates a default placement strategy whereby it places each tier of the application onto a particular machine. Purposefully we start off with a low number of machines (2) and gradually increase the load to identify the right number of resources required. This approach helps to avoid a situation where we need to reduce the number of machines (M − 1) on each iteration. The algorithm makes a call onto the Mean Value Analysis (MVA) algorithm. The MVA returns the utilization of each tier which can be used to find the bottleneck machine (the machine with the highest utilization). The tier present in that machine is then replicated and placed in a new machine which is introduced in that iteration. In this manner, the iteration continues until the total number of machines equal the given maximum machines.

Algorithm 1: Response Time Analysis (RTA)

```
Input:       Ld, Predicted Workload
             Hw, Total Machines available
             SD, Service Demand for the job classes
             Z, Think Time
Output:
             Response Time R Vector of response times for all job classes

begin
 // start with one machine per tier (2 in this case)
 M = 2;
 while M <= Hw do
      // Get response time and server utilization by running MVA on
      // analytical model presented in [20]
      [R, U] = MVA (SD, Ld, Z);
      i = maxUtil (U); // Get the index of the bottleneck tier
      // Add a machine and replicate tier i on it to balance the load
      M = M + 1
      M = i
done
end
```

11.3.3 Optimizing Resource Provisioning

To optimize resource usage and minimize idle resources, the best way would be to define a time interval and change resources as many times as possible as workload changes. In the limit this interval could be made infinitesimally small and resources changed continuously, however, as noted earlier such an extreme solution is not feasible. The intuition therefore is to identify the right number of time intervals in which to make these adjustments with the requirement that the time interval is neither too small nor too large. Our solution works on the principle of receding horizon control also known as look-ahead optimization [81].

This form of controller iteratively solves an optimization problem, Cost^{opt} starting from t0, over a predefined horizon (t = 1...N) taking into account current and future constraints. Once a feasible sequence is found, only the first input in the sequence is applied and the rest are discarded. Effectively, the optimization search results in the construction of a tree with branching factor K and N + 1 levels. Here K is the total number of finite input choices. Formally, at time t0, given state x_{t0}

$$\text{Cost}^{\text{opt}} = \min \{\text{Cost}(\{xt\}, \{ut\})\} \text{ where } \{\} \text{ denotes a set}$$

$$x_{t+1} = f(x_t, u_t), t = t_0 \dots t_{N-1}$$

$$u_t \in U \text{ finite input choices}$$

$$X_N \text{ is the set of final goal states}$$

$$Cost\ (\{xt\}, \{ut\}) = (\sum_{t=t0+1}^{t=t0+N-1} J\big(x(t), y(t)\big), where\ J\ is\ a\ utility\ function$$

Sequences $\{u_t\} = \{u_0, \cdots, u_{N-1}\}$ and $\{x_t\} = \{x_0, \cdots, x_{N-1}\}$ are the feasible input sequence and the resulting states that trace a path from the root to the lead node in this search tree such that the net cost across the sum of all branches is minimum and the leaf node is closest to the final destination state. Given that this method needs finite input choices, we use a finite range of machines that can be increased, decreased, or kept the same. The next challenge is the choice of the look-ahead period. A small look-ahead period will neglect trends, while a very large period will increase computational complexity and lead to a larger prediction error, which will yield any control decision ineffective. Thus, the number of look-ahead periods needs to balance out the different tradeoffs.

Our algorithm uses the receding horizon control and iterates over the number of look-ahead steps and calculates the cumulative costs. For every future time step, it computes the cost of selecting each possible resource allocation. To compute the cost of a particular allocation, it uses Algorithm 1 to compute the estimated response time for that particular machine configuration. Once the response time is calculated, it is used to calculate the cost of the allocation which is a combination of how far the estimated response time is from the SLA bounds, cost of leasing additional machines and also a cost of re-configuration.

$$W_t = \beta \times W_{t-1} + \gamma \times W_{t-2} + (1 - \beta - \gamma) \times W_{t-3} \quad (2)$$
$$M_t = M_{t-1} + u_t \quad (3)$$
$$R_t = \text{ResponseTimeAnalysis}(W_t, M_t, SD, Z) \quad (4)$$

The cost of reconfiguration is computed based on the number of machines that need to be updated. Obviously re-configuration will incur some costs and thus the algorithm will try to reduce the amount of reconfiguration. Each of these cost components will have weights attached to them which may be varied depending on the type of application and its requirements. Applications are required to specify which factors are more important to them, and our auto-scaling algorithm will honor these specifications in making the decisions.

11.4 Experimental Evaluation

This section presents a subset of results evaluating our look-ahead algorithm for auto-scaling in cloud environments [83]. We first show how the algorithm determines the number of resources to be allocated in a just-in-time manner so that the overall cost is minimized. Next, the effects of different cost weightings are studied. This study is important since different applications may impose different weighting combinations. The data used in this study is acquired from the 1998 soccer world cup web site shown in Figure 28. We use the number of customers visiting that site as an indication of the amount of workload that typically can be experienced by such a globally popular topic.

11.4.1 Just-in-time Resource Allocation

To evaluate the strength of our just-in-time resource allocation, we have used a cost function shown in Equation 5 comprising three components. Recall that the three components of the cost function refer individually to the penalty for violation of SLA bounds, cost of leasing a machine, and cost of reconfiguring the application when machines are either leased or released. Each of these components has a weight attached to it and the system can be made to always minimize a certain component by increasing the attached weight to it to an arbitrary high value. Table 2 describes the components of the cost function.

$$\text{Cost} = W_r \times (R_{sla} - R) + W_c \times M_k + W_f \times (M_k - M_{k-1}) \quad (5)$$

Component	Description	Unit
W_r	Penalty for SLA violation	\$/sec
W_c	Cost of leasing a machine per hour	\$/machine
W_f	Cost of reconfiguration	\$/machine
R_{sla}	SLA for the observed response time	sec
R	Maximum response time for an application	sec
M_k	Number of machines used in the k^{th} interval	Numeric
M_{k-1}	Number of machines used in the $(k-1)^{th}$ interval	Numeric

Table 2: Components of the Cost Equation

For this experiment, the weights on each component of the cost function are the same, which means all factors are equally important. Figure 30 shows how the look-ahead algorithm determines changes in the resources required as the incoming load changes. The computation is done on the basis of predicted workload which is done with the help of the ARMA filter given in Equation 1. Figure 30 clearly shows that the base resources required are 2 machines and it increases to 3 or 4 when the load is increased. The prediction of the look-ahead algorithm based on a selected number of time intervals closely matches the incoming load. It prescribes a resource increase whenever there is high load and fewer resources when there is a lower load. Thus Figure 30 shows the effectiveness of the look-ahead algorithm and how it can save cost while also assuring that the performance of the application is assured.

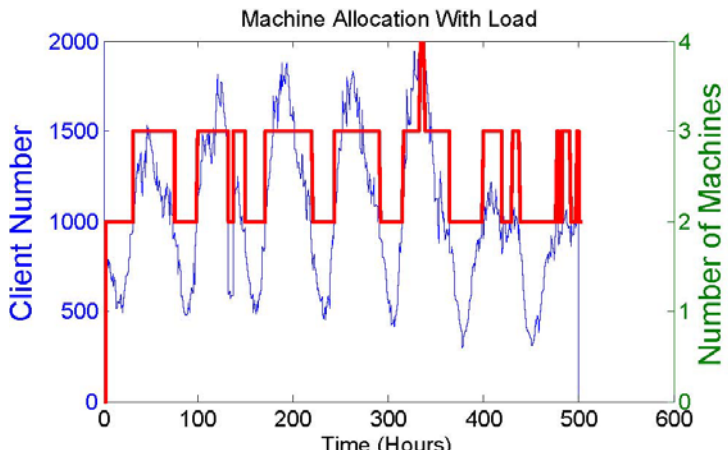


Figure 30: Just-in-time Resource Allocation with Changing Load

11.4.2 Resource Usage under Different Cost Priorities

The results in this section demonstrate the allocation/deallocation of resources stemming from using different cost ratios among the three competing factors in the cost function of Equation 5. The resource allocation determined by our algorithm in the different time intervals will depend upon the weights assigned to the various components of the cost function. The rest of the section studies the different trends of resource allocation and how they are influenced by the varying weights of the cost function.

SLA violation against Resource Cost: We first show the results when considering the effect of SLA violation against cost of resources. The ratio of the cost of SLA violation against the cost of machines is varied while the application reconfiguration cost is assumed to be zero. We assume that the application can be easily reconfigured with varying machines. The ratio of SLA penalty to machine cost is varied from 4:1 (which means SLA violation is higher priority than cost of the machine) to 1:13 (which means that the machine cost is higher priority than SLA violation).

Figure 31 shows how the resources are allocated every hour over the entire time period. The corresponding cost values are also shown in the bottom graph for each of these figures. The intervals over which there is SLA violations are also shown. The algorithm always tries to keep the cost to a minimum. It is seen that there is significant difference in resource allocation between the different configurations. An application with high SLA violation penalty has stronger performance assurance whereas one with low SLA penalty has lesser performance assurance. The priorities of the application determine the difference in resource allocation. For a low performance assurance and high machine cost, the number of machines used is only two over the entire time interval. The cost of machines exceeds the cost of SLA violations and such a configuration will have to tolerate a number of SLA violations

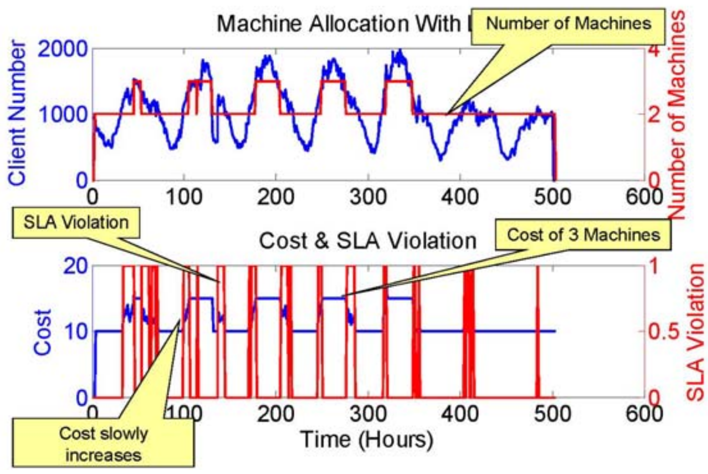


Figure 31: Resource Allocation for Low SLA Violation Cost and High Machine Cost (1:13:0)

On the contrary, Figure 32 shows how the algorithm supports an application with a high SLA violation cost. For the highly assured application of Figure 32, there is much variation in resource usages with a number of intervals having 3 machines and also some having 4 machines. Here the priority is in assuring performance and the cost of machines is much lower.

Finally, Figure 33 shows the distribution of the number of machines required for a variety of systems ranging from highly assured systems (ratio of SLA violation penalty to machine cost being 4 : 1) to very weakly assured systems (ratio of SLA violation penalty to machine cost being greater than 1 : 13). In this figure, each point on the X-axis is a ratio of the cost of an SLA violation to the cost of a machine. The Y-axis plots the number of intervals in which each type of machine is used. For example, for the point corresponding to cost ratio of 1:4, 359 intervals use 2 machines and the other 143 intervals use 3 machines. The ratio of SLA violation cost-to-machine cost increases as we move further down the X-axis. The figure shows more the use of 3 machines than 2 machines as we move to the right. This outcome is because the relative cost of machines decreases to the right and the penalty of SLA violation increases.

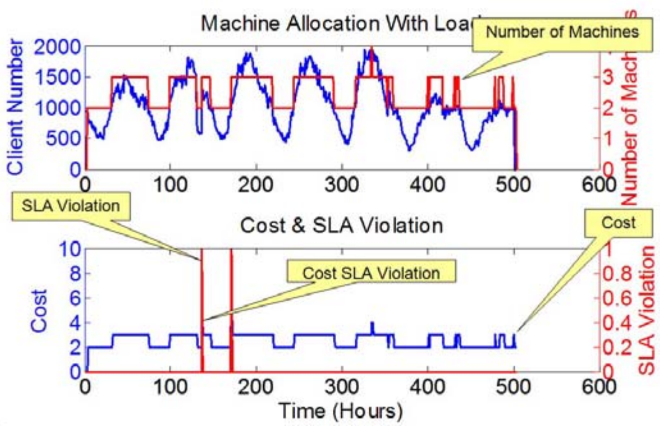


Figure 32: Resource Allocation for High SLA Violation Cost (1:4:0)

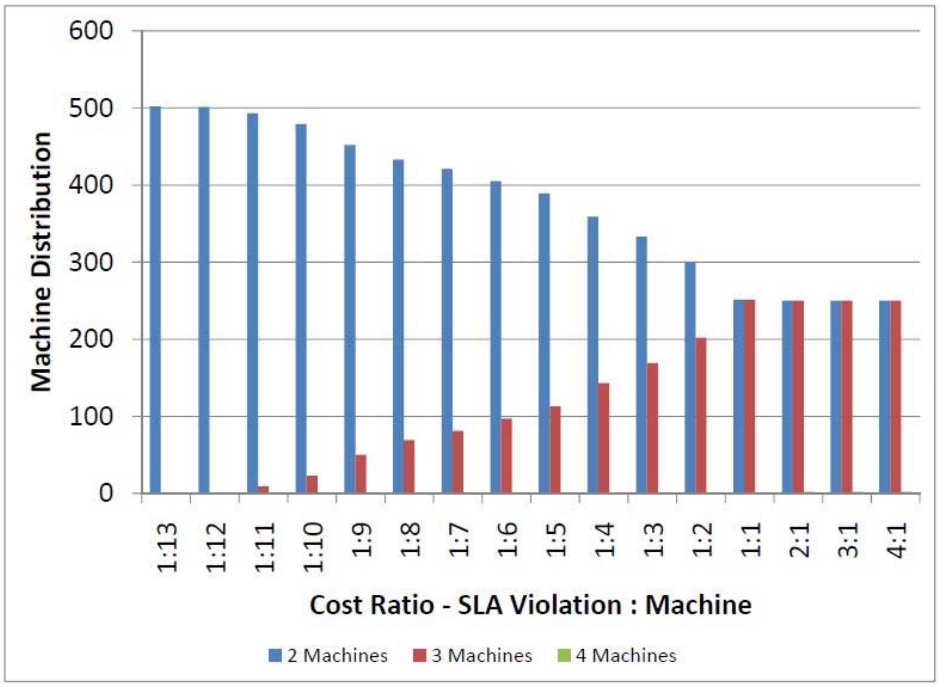


Figure 33: Resource Allocation for Variety of Systems

Including the Cost of Reconfiguration: Figure 32 showed how resource allocation is done when there is high SLA violation cost compared to machine cost. For this configuration, in every interval, the mean response time is below the SLA bound and the machines are allocated whenever they are needed. A machine is released again since there is cost of machine but only making sure that the SLA is maintained. When there is a cost of reconfiguration introduced, the algorithm will resist the changing of resources. This phenomenon can be related to inertia in physical bodies. Inertia resists changes to its current physical condition such as a body in rest resists movement while a body in motion resists slowing down. Thus, the cost of reconfiguration will similarly resist the dynamic nature of resource allocation.

The higher the cost, the higher will be its resistance to the changes. This cost is expressed as the third component of Equation 5. The weight W_f represents the level of inertia and it is multiplied by the change level which is the number of machines allocated or released. Initially when a small amount of reconfiguration cost is introduced, it does not affect much as shown in Figure 35. The resource allocation is similar to Figure 32. There are small deviations, where the spikes in resource changes are a little wider in Figure 34 than in Figure 32. This is due to the inertia in change introduced due to some cost associated with change.

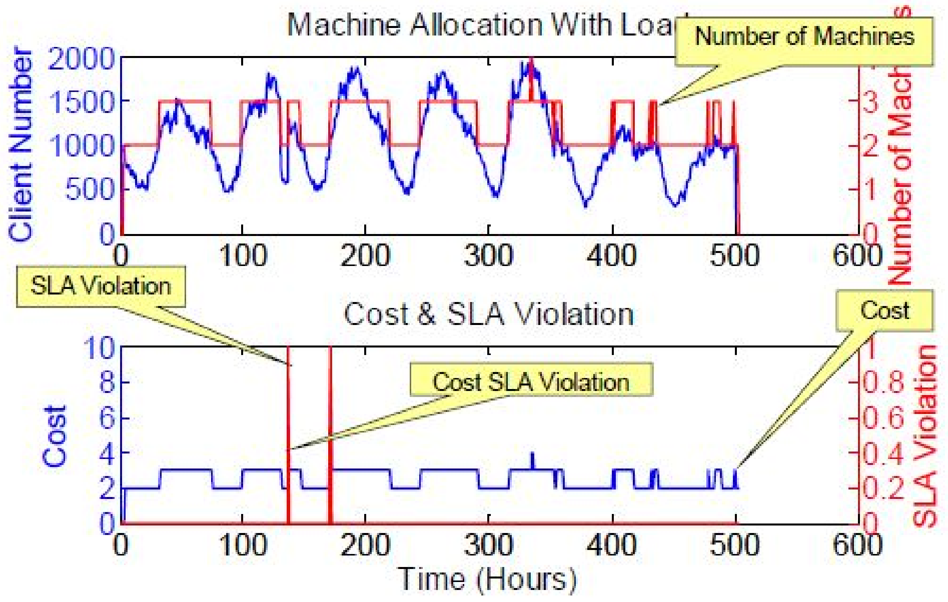


Figure 34: High SLA Violation with Low Reconfiguration Cost

The effect of the cost of reconfiguration is more pronounced when it is prioritized slightly higher. Figure 35 shows a distinct change in resource allocation over the hourly intervals compared to Figure 32 or Figure 34. In Figure 35, the number of machines increases to 3 around the 40th hour and remains steady. Somewhere around the 350th hour it increases to 4 machines since the workload increased at that time. Subsequent to that, the workload decreased but the machines were never released since the cost of reconfiguration is considered much higher compared to the cost of machines. The changes of the machines around 40 and 350 hours was warranted because of the high SLA violation cost and the machines were never released even though the workload lessened since the cost of reconfiguration was higher.

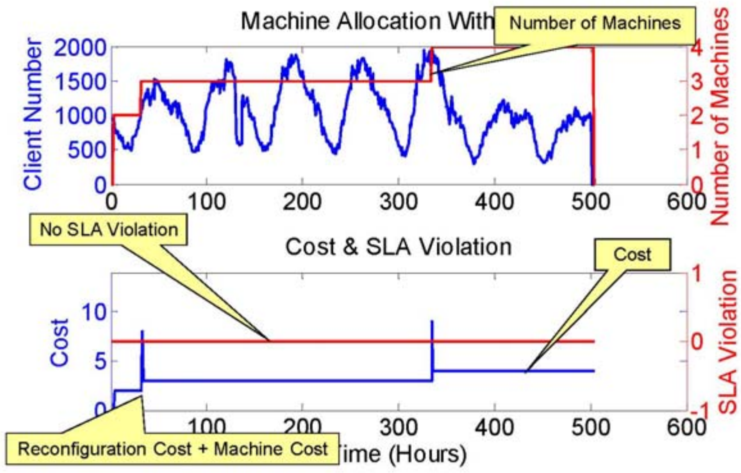


Figure 35: High SLA Violation with Medium Reconfiguration Cost

This behavior of resisting change is further pronounced in Figure 36 where there is even higher cost of reconfiguration. Here again there is an increase of machines to 3 at around the 40 hour mark and the machine is never released. The change to 4 machines which was seen in Figure 35 does not occur here because the cost of reconfiguration is much higher than the cost of SLA violation. Thus even though there is SLA violation, it is only of a short duration (the peak workload around 350 hours) and is of lesser cost than the cost of changing resources. That the SLA violation near 300 hours was of a short duration can be understood from Figure 34 where there is a very short spike of machine allocation to 4 around that time. When the cost of reconfiguration becomes high, the look-ahead algorithm decides not to expend the extra cost of reconfiguration to cover up that short SLA violation.

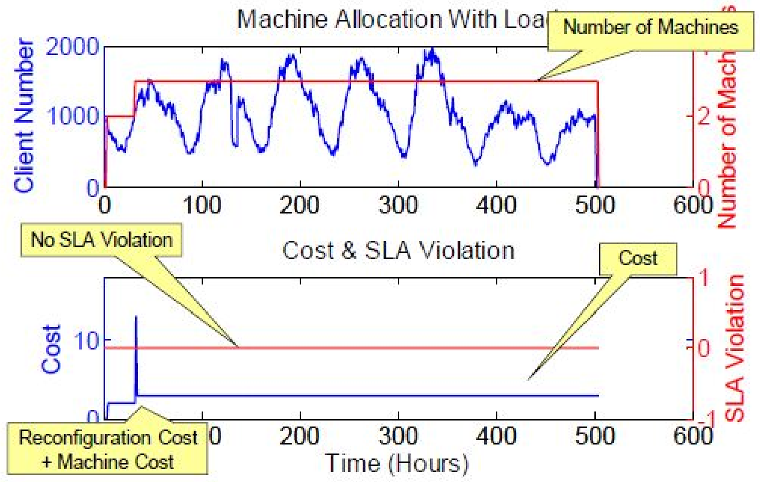


Figure 36: High SLA Violation with High Reconfiguration Cost

11.5 Concluding Remarks

Auto-scaling of resources helps cloud service providers operating modern day data centers to support a maximal number of customers while assuring customer QoS requirements in accordance with service level agreements, and keeping cost of using resources low for customers. However, current auto-scaling mechanisms require user input and programming of APIs to adjust resources as workloads change. Reactive scaling of resources imposes performance overheads while also making the programming of the cloud infrastructure tedious. To address these problems, this report describes a look-ahead resource allocation algorithm based on model-predictive control which predicts future workload based on a limited horizon and adjusts resources allocated to users ahead-of-time. Empirical results evaluating our approach show significant benefits both to cloud users and providers. The work presented demonstrates the feasibility of our approach in the context of the small number of machines used. Our future work will explore the scalability of our algorithms in the context of modern day workloads and large numbers of resources, which are typical of contemporary applications.

12. Runtime Infrastructure for Deployment and Configuration

This section focuses on the runtime infrastructure that enables the deployment and validation of DoD system artifacts in the runtime execution environment. Predictable performance of these capabilities is important for assuring QoS properties of DoD systems.

12.1 Introduction

Component-based middleware, such as the Lightweight CORBA Component Model, are increasingly used to implement large-scale distributed, real-time and embedded (DRE) systems. In addition to supporting the quality of service (QoS) requirements of individual DRE systems, component technologies must also support bounded latencies when effecting deployment changes to DRE systems in response to changing environmental conditions and operational requirements.

Component-based software engineering techniques are increasingly applied to develop large-scale distributed real-time and embedded (DRE) systems, such as air-traffic management, shipboard computing environments, and distributed sensor webs. These domains are often characterized as "open" since applications in these domains must contend not only with changing environmental conditions (such as changing power levels, operational nodes, or network status), but also evolving operational requirements and mission objectives [84].

To adapt to changing environments and operational requirements, it may be necessary to change the deployment and configuration characteristics of these DRE systems at runtime. Examples of potential adaptations include deployment or teardown of individual component instances, changing connection configuration, or altering QoS properties in the target component runtime. As a result of stringent quality of service (QoS) requirements in these domains, it is important that any changes to DRE system deployment and configuration occur as quickly and predictably as possible. That is, DRE systems expect short and bounded deployment latencies.

Not only are timely and dependable runtime deployment and configuration changes essential in DRE systems, even initial application startup time can be an important metric. For example, in extremely energy-constrained systems, such as distributed sensor networks, a common power saving strategy may involve completely deactivating field hardware and periodically restarting it to take new measurements or activate actuators. In such environments, deployments must be fast and time-bounded.

To support these requirements, the efficiency and QoS provided by the deployment infrastructure should be considered alongside the component middleware used to develop DRE systems. Standards, such as the Object Management Group (OMG) Deployment and Configuration (D&C) specification [85] for component-based applications, have emerged in recent years. The OMG D&C specification provides comprehensive development, packaging, and deployment frameworks for a wide range of component middleware. Although originally developed for the CORBA Component Model (CCM), the OMG D&C specification is defined via a UML meta-modeling that is applicable to many other component models.

In the OMG D&C specification, deployment instructions are delivered to the deployment infrastructure via a component deployment plan (CDP), which contains the complete set of deployment and configuration information for component instances and their associated connection information. During DRE system initialization, such information must be parsed, components deployed on the nodes, and the system activated in a timely and predictable manner. We refer to the timeliness of the deployment infrastructure as the “deployment latency,” which includes the time starting when a CDP is provided to the deployment infrastructure to the time at which all deployment instructions have been executed and the system activated.

This section motivates and describes architectural enhancements we made to the OMG D&C specification to achieve predictable deployment latencies for large-scale DRE systems. Our solution is called the Locality-Enhanced Deployment and Configuration Engine (LE-DAnCE), which extends our earlier Deployment and Configuration Engine (DAnCE) [86]. We developed DAnCE with the sole aim of cleanly separating concerns defined by the OMG D&C specification and demonstrating its feasibility. After applying DAnCE to a range of representative DRE systems, however, we found the lack of appropriate optimizations and architectural limitations of the OMG D&C specification yielded performance bottlenecks that adversely impacted deployment latencies. Moreover, these performance bottlenecks stemmed from more than just limitations with the original DAnCE implementation, but involve inherent architectural limitations with the OMG D&C specification itself.

12.2 Impediments to Predictable Deployment Latency

We expose key sources of overhead that impact deployment latencies in DRE systems and pinpoint the architectural limitations in the D&C specification that exacerbate these overheads. The OMG D&C specification, whose architecture is shown in Figure 37 provides standard interchange formats for metadata used throughout the component application development lifecycle, as well as run time interfaces used for packaging and planning. Below we focus on the interfaces, metadata, and architecture used for runtime deployment and configuration. The major sources of latency overhead stem from multiple complexities in the OMG D&C standard, including the processing of deployment metadata from disk in XML format and an architectural ambiguity in the runtime infrastructure that encourages sub-optimal implementations.

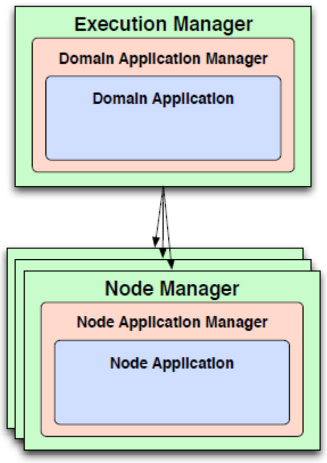


Figure 37: OMG D&C Architectural Overview

Challenge 1: Parsing Deployment Plans. Component application deployments for OMG D&C are described by a data structure that contains all the relevant configuration metadata for the component instances, their mappings to individual nodes, and any connection information required. This CDP is serialized on disk in a XML file whose structure is described by an XML Schema defined by the OMG D&C standard. This XML document format for CDP files presents significant advantages by providing a simple interchange format between modeling tools, is easy to generate and manipulate using widely available XML modules for popular programming languages, and enables simple modification and data mining by text processing tools, such as perl, grep, sed, and awk.

Processing these CDP files during deployment and even runtime, however, can lead to substantial deployment latency costs [87]. This increased latency stems from the following sources: XML CDP file sizes grow substantially as the number of component instances and connections in the deployment increases, which causes significant I/O overhead to load the plan into memory and to validate the structure against the schema to ensure that it is well-formed.

The XML document format cannot be directly used by the deployment infrastructure, so it must first be converted into the native OMG Interface Definition Language (IDL) format used by the runtime interfaces of the deployment framework. In many enterprise DRE systems, component deployments that number in the thousands are not uncommon, and component instances in these domains will exhibit a high degree of connectivity. Given the structure of these common DRE systems, both these factors contribute to large plans. While the above latency source is most immediately applicable to initial application deployment, it can also present a significant problem during potential redeployment activities at application runtime that involve significant changes to the application configuration. While CDP files that represent redeployment or reconfiguration instructions may not be as large as for the initial deployment, the responsiveness of the deployment infrastructure during these activities is even more important to ensure that the application continues to meet its stringent QoS and end-to-end deadlines during online modifications.

Challenge 2: Serialized Execution of Deployment Actions. The complexities presented earlier in this section involve the serial (non-parallel) execution of deployment tasks. The related sources of latency in DAnCE exist at both the global and node level. At the global level, this lack of parallelism results from the underlying CORBA transport used by DAnCE. The lack of parallelism at the local level, however, results from the lack of specificity in terms of the interface of the D&C implementation with the target component model that is contained in the D&C specification. The D&C deployment process enables global entities to divide the deployment process into a number of node-specific subtasks. Each subtask is dispatched to individual nodes using a single remote invocation, with any data produced by the nodes passed back to the global entities via "out" parameters that are part of the operation signature described in IDL. Due to the synchronous nature of the CORBA messaging protocol used to implement DAnCE, the conventional approach is to dispatch these sub-tasks serially to each node. This approach is simple to implement, in contrast to the complexity of using the CORBA asynchronous method invocation (AMI) mechanism [88].

To minimize initial implementation complexity, we used synchronous invocation (admittedly shortsighted) in DAnCE. This global synchronicity did not cause problems for relatively small deployments (less than 100 components). However, as the number of both nodes and instances assigned to the nodes begin to scale up, the global/local serialization will impose a substantial cost in deployment latency.

This serialization problem is not limited only to the global/local task dispatching, but also exists in the node-specific portion of the infrastructure. The D&C specification provides no guidance in terms of how the Node-Application should interface with the target component model (in this case, CCM), instead leaving such an interface as an implementation detail. Early versions of DAnCE directly instantiated the CCM containers and components directly in the address space of the Node Application. To alleviate the resulting tedious and error-prone deployment logic, we later separated the CCM container into a separate process. In DAnCE, the D&C architecture was implemented using three processes, as shown in Figure 38.

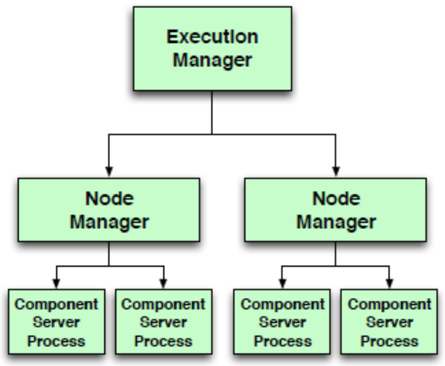


Figure 38: Simplified DAnCE Architecture

The Execution Manager and Node Manager processes instantiate their associated Application Manager and Application instances in their address space. When the Node Application installs the concrete component instances it spawns one, or more, separate component server processes as needed. The component server processes use an interface derived from an older version of the CCM specification that allows the Node Application to individually instantiate containers and component instances. This approach is similar to that taken by CARDAMOM (cardamom.objectweb.org), which is another CCM implementation tailored for enterprise DRE systems, such as air-traffic management systems.

While the DAnCE architecture shown in Figure 38 improved upon the original implementation that collocated all CCM entities in Node Application address space, it was still problematic with respect to parallelization. Rather than performing only some processing and delegating the remainder of the concrete deployment logic to the component server process, the DAnCE Node Application implementation instead integrates all logic necessary for installing, configuring, and connecting instances directly, as shown in Figure 39.

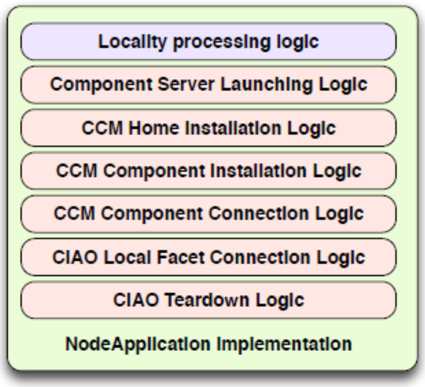

Locality processing logic
Component Server Launching Logic
CCM Home Installation Logic
CCM Component Installation Logic
CCM Component Connection Logic
CIAO Local Facet Connection Logic
CIAO Teardown Logic
NodeApplication Implementation

Figure 39: DAnCE Node Application Implementation

This tight integration made it hard to optimize the node-level installation process for the following reasons:

- The amount of data shared by the generic deployment logic (the portion of the Node Application implementation that interprets the plan) and the specific deployment logic (the portion which has specific knowledge of how to manipulate components) made it hard to parallelize their installation since that data must be modified during installation.

- Since groups of components installed to separate component servers can be considered separate deployment sub-tasks, these groupings could be also parallelized.

12.3 Overcoming Deployment Latency Bottlenecks in Le-Dance

12.3.1 Improving Runtime Plan Processing

There are two approaches to resolving the challenge of XML:

1. **Optimize the XML to IDL processing capability.** DAnCE uses a vocabulary-specific XML data binding [89] tool called the XML Schema Compiler (XSC). XSC reads D&C XML schemas and generates a C++-based interface to XML documents built atop the Document

Object Model (DOM) XML programming API. In general, DOM is a time/space-intensive approach since the entire document must first be processed to fully construct a tree-like representation of the document before the XML-to-IDL translation process can occur.

An alternative is to use the Simple API for XML (SAX), which uses an event-based processing model to process XML files as they are read from disk. While a SAX-based parser would reduce the time/space spent building the in-memory representation of the XML document, the performance gains may be too small to invest the substantial development time required to re-factor the DAnCE configuration handlers, which serve as a bridge between the XSC-generated code and IDL. In particular, a SAX-based approach would still require a substantial amount of runtime text-based processing. Moreover, CDP files have substantial amounts of internal cross-referencing, which would require the entire document be processed before any actual XML-to-IDL conversion could occur.

2. **Pre-process the XML files for latency-critical deployments.** This optimization approach (used by LE-DAnCE) is accomplished via a tool we developed that leverages the existing DOM-based XML-to-IDL conversion handlers in DAnCE to (1) convert the CDP into its runtime IDL representation and (2) serialize the result to disk using the Common Data Representation (CDR) binary format defined by the CORBA specification. This platform-independent binary format used to store the CDP on disk is the same format used to transmit the plan over the network at runtime. The advantage of this approach is that it leverages the heavily optimized de-serialization handlers provided by the underlying CORBA implementation (TAO) to create an in-memory representation of the CDP data structure from the on-disk binary stream.

12.3.2 Parallelizing Deployment Activity

To support parallelized dispatch of deployment activity at the node level, we enhanced the OMG D&C standard by adding a Locality Manager to LE-DAnCE. The Locality-Manager unifies all three deployment roles, and functions as a replacement for the component server in Figure 38. An overview of LE-DAnCE's Locality-Manager appears in [90].

The LE-DAnCE node-level architecture (e.g.,Node Manager, Node Application Manager, and Node Application) now functions as a node-constrained version of the global portion of the OMG D&C architecture. Rather than having the Node Application directly causing the installation of concrete component instances, this responsibility is now entirely delegated to Locality Manager instances. The node-level infrastructure performs a second "split" of the plan it receives from the global level by grouping component instances into one or more component servers. The Node Application then spawns a number of Locality Manager processes and delegates these "process-constrained" (i.e., containing only components and connections apropos to a single process) plans to each process in parallel.

Unlike the previous DAnCE Node Application implementation, the LE-DAnCE Locality Manager functions as a generic application server that maintains a strict separation of concerns between the general deployment logic required to analyze the plan and the specific deployment logic required to actually install and manage the lifecycle of concrete component middleware instances. This separation is achieved using entities called Instance Installation Handlers, which provide a well-defined interface for managing the life-cycle of a component instance, including installation, removal, connection, disconnection, and activation. Installation Handlers are also used in the context of the Node Application to manage the life-cycle of Locality Manager processes.

Figure 40 shows the startup process for a Locality Manager instance. During the start launch phase of deployment, an Installation Handler hosted in the Node Application spawns a Locality Manager process and handles the initial handshake to provide configuration information. The Node Application then instructs the Locality Manager to begin deployment by invoking preparePlan() and startLaunch(). During this process, the Locality Manager will examine the plan to determine what instance types must be installed (e.g., container, component, or home). After loading the appropriate Installation Handlers will delegate the actual installation process for these instances via the install_instance() method on the Installation Handler.

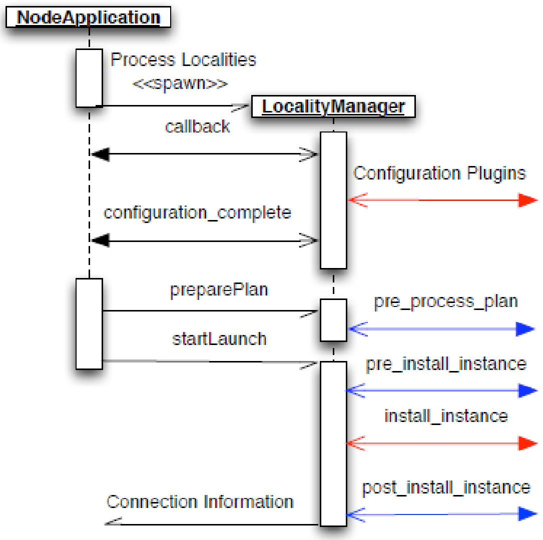


Figure 40: Locality Manager Startup Sequence

The new LE-DAnCE Locality Manager and Installation Handlers make it substantially easier to parallelize than in DAnCE. Parallelism in both the Locality Manager and Node Application is achieved using an entity called the Deployment Scheduler, which is shown in Figure 41. The Deployment Scheduler combines the Command pattern [91] and the Active Object pattern [92]. Individual deployment actions (e.g., instance installation, instance connection, etc.) are encased inside an Action object, along with any required metadata. Each individual deployment action is an invocation of a method on an Installation Handler, so these actions need not be re-written for each potential deployment target. Error handling and logging logic is also fully contained within individual actions, further simplifying the Locality-Manager.

Individual actions (e.g., install a component or create a connection) are scheduled for execution by a configurable threadpool, which can provide user-selected single-threaded or multi-threaded behavior, depending on the requirements of the application. This thread pool could also be used to implement more sophisticated scheduling behavior. For example, it might be desirable to implement a priority-based scheduling algorithm that dynamically reorders the installation of component instances based on metadata present in the plan.

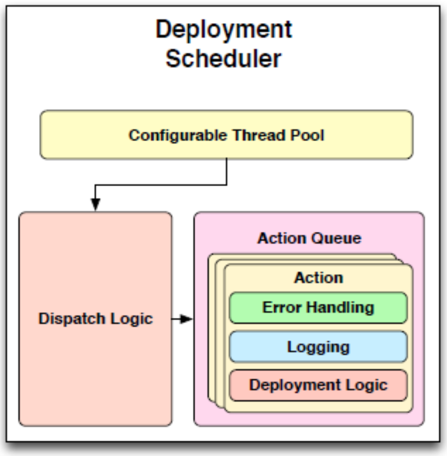


Figure 41: DAnCE Deployment Scheduler

During deployment, the Locality Manager determines which actions to perform during each particular phase and creates one Action object for each instruction. These actions are then passed to the deployment scheduler for execution while the main thread of control waits for a completion signal from the Deployment Scheduler. Upon completion, the Locality-Manager reaps either return values or error codes from the completed actions and completes the deployment phase.

To provide parallelism between Locality Manager instances on the same node the LE-DAnCE Deployment Scheduler is also used in the implementation of the Node Application, along with an Installation Handler for Locality Manager processes. Using the Deployment Scheduler at this level also helps to overcome a significant source of latency whilst conducting node-level deployments. Spawning Locality Manager instances can take a significant amount of time compared to the deployment time required for component instances, so parallelizing this process can achieve significant latency savings when application deployments have many Locality-Manager processes per node.

12.4 Concluding Remarks

This section described the OMG Deployment and Configuration (D&C) specification for component-based applications and explored sources of deployment latency overhead that degraded the responsiveness of the Deployment And Configuration Engine (DAnCE),which is an open-source implementation of the D&C specification. We then explained how our Locality-Enhanced Deployment and Configuration Engine (LE-DAnCE) enhanced DAnCE to alleviate key sources of deployment latency overhead associated with XML preprocessing and LocalityManager architecture. The effectiveness of the LE-DAnCE LocalityManager architecture was then empirically evaluated [87] by (1) deploying a number of high component-density applications to demonstrate the performance of the toolchain as the number of components grows and (2) measuring the predictability of these performance results by repeatedly deploying the same setup on a 1,000 component deployment.

The following lessons were learned conducting this research: Split Plan process incurs significant deployment latency. The results showed that the plan preparation phase of deployment is a large source of deployment latency, due in large part to inefficiency in the LE-DAnCE "split plan" algorithm [87]. To alleviate this inefficiency our future work will determine if this algorithm can further be optimized or investigate ways that the plan can be split before deployment to reduce runtime deployment latency.

The startLaunch operation is a significant source of jitter. The start launch phase of deployment produces the largest amount of jitter in the LE-DAnCE deployment process. Prior experiments [93] conducted on DAnCE showed this jitter stemmed from the dynamic loading of component implementations at runtime and can be alleviated by directly compiling component implementations and plan metadata into the deployment infrastructure. While this approach reduces jitter and latency, it is also invasive into the D&C implementation, hard to maintain, and removes much of the flexibility from the D&C toolchain. Our future work is exploring more flexible ways to reduce this jitter via work that builds on these previous efforts.

CIAO and LE-DAnCE are open-source software and all work described is available in the latest version which can be obtained from download.dre.vanderbilt.edu.

13. Conclusions

13.1 Summary

Our prior work in the Air Force's Global Information Grid (GIG), Army's Future Combat Systems (FCS) program, and the Navy's DDG 1000 program activities motivated a challenging problem facing developers of large-scale and layered DoD software-intensive net-centric systems: *how to discover, measure, and rectify structural, integration, and/or performance problems early in the system's lifecycle (e.g., in the architecture and design phases), as opposed to the final system integration phase, when mistakes are much harder and more costly to fix***.** We observed that the bulk of today's software technologies and validation/verification techniques are designed to develop and analyze relatively small-scale systems, where the set of tasks that will run in the system and their requirements for system resources are known in advance. Unfortunately, these technologies are poorly suited for large-scale DoD systems, where it is not possible to know the entire set of application tasks that will run on the system, the loads they will impose on system resources in response to a dynamically changing environment, or the order in which the tasks will execute. Moreover, even in today's smaller-scale DoD systems where load is known in advance, there is often little confidence that system quality of service (QoS) requirements will be met in the deployment phase due to the complexities of analyzing complex systems built atop commercial-off-the-shelf (COTS) hardware and software components.

Net-centric DoD systems must provide QoS support to process the right data in the right place at the right time over a networked grid of computers. Some QoS properties required by these DoD systems include the low latency and jitter as expected in conventional real-time and embedded systems, and high throughput, scalability, and reliability as expected in conventional enterprise distributed systems. Achieving this combination of QoS capabilities in DoD systems is hard, particularly when the systems are developed using COTS hardware/software components.

The three year project called "System Execution Modeling Technologies for Large-scale Net-centric DoD Systems" conducted research to address these challenges, and resulted in practical artifacts as part of a tool suite called "GUTS: GEMS Utilization Test Suite." The R&D we conducted sought solutions along three dimensions:

- Design-time solutions – where we conducted R&D in model-based algorithms and technologies;
- Deployment-time solutions – where we conducted R&D on a variety of deployment-related optimization solutions, often based on heuristics;
- Run-time solutions – where we conducted R&D on runtime architectures to support the design- and deployment-time decisions.

During the course of the project, we focused primarily on emerging platforms and technologies of interest to DoD including cloud environments and mobile platforms, all the while focusing on validating and assuring QoS properties as well as other physical properties, such as energy and power consumed, and environmental issues. All the artifacts we have developed are available in open source. In particular, the following URLs summarize the availability of these tools:

- CoSMIC/CUTS SEM tools that are part of GUTS are available from http://www.dre.vanderbilt.edu/cosmic
- GEMS is available from the Eclipse modeling project pages at http://www.eclipse.org/gmt/gems/
- ASCENT and related technologies are available from http://code.google.com/p/ascent-design-studio/
- All middleware sources are available from http://www.dre.vanderbilt.edu/downloads

13.2 Future Work

Our work to date has made initial forays into the world of resource management and deployment optimizations for tactical cloud environments that will increasingly be essential to realize increasingly complex DoD systems and operations. With the increasing proliferation of mobile devices including smart phones, and significant challenges stemming from security issues, we believe there are ample opportunities to conduct additional R&D. In particular, we will leverage our new DURIP equipment award to set up a smart phone-based cloud environment, and seek solutions to emerging problems in this space.

13.3 Representative Publications

This project resulted in multiple publications. Below we list the relevant and major publications stemming from the effort that covers the entire range of our contributions.

1. J. White, D. C. Schmidt, D. Benavides, P. Trinidad, and A. Ruiz-Cortez. "Automated Diagnosis of Product-line Configuration Errors in Feature Models," In *Proceedings of the Software Product Lines Conference (SPLC)*, Limerick, Ireland, Sept. 2008.
2. J. White, D. Benavides, B. Dougherty, and D. Schmidt. "Automated Reasoning for Multi-step Configuration Problems," In Proceedings of the Software Product Lines Conference (SPLC), San Francisco, USA, Aug. 2009.
3. J. White, B. Dougherty, and D. Schmidt. "Selecting highly optimal architectural feature sets with Filtered Cartesian Flattening," The Journal of Systems & Software, 82(8):1268–1284, 2009.
4. J. White, B Dougherty, C. Thompson, and D. C. Schmidt. "ScatterD: Spatial Deployment Optimization with Hybrid Heuristic / Evolutionary Algorithms," Proceedings of the Conference on Evolutionary Algorithms, 2010; Also to Appear in ACM Transactions on Autonomous and Adaptive Systems Special Issue on Spatial.
5. A. Shah, K. Ann, A. Gokhale, and J. White, "Maximizing Service Uptime of Smartphone-based Distributed Real-time and Embedded Systems," Proceedings of the

IEEE 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2011, Irvine, CA, USA, pp. 3—10.

6. N. Roy, A. Dubey, and A. Gokhale, “Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting,” *Proceedings of the 4th IEEE International Conference on Cloud Computing (CLOUD ’11)*, Washington, D.C, USA, July 4—9, 2011, pp. 500—507.

7. W. Otte, A. Gokhale, and D. C. Schmidt, “Predictable Deployment in Component-based Enterprise Distributed Real-time and Embedded Systems,” *Proceedings of the ACM Component-based Software Engineering*, Boulder, CO, June 2011, pp. 21—30.

8. J. White, B. Dougherty, C. Thompson, and D. C. Schmidt, “ScatterD: Spatial Deployment Optimization with Hybrid Heuristic,” in *Evolutionary Algorithms*, 2010.

9. J. White, B. Doughtery, and D. C. Schmidt, “Ascent: An algorithmic technique for designing hardware and software in tandem,” *IEEE Transactions on Software Engineering*, pp. 838–851, 2010.

10. W. Otte, D. Schmidt, and A. Gokhale, “Towards an Adaptive Deployment and Configuration Framework for Component-based Distributed Systems,” in *9th Workshop on Adaptive and Reflective Middleware (ARM ’10)*, Begaluru, India, 2010.

14. References

[1] R. P. Gabriel, L. Northrop, D. C. Schmidt, and K. Sullivan, “Ultra-large-scale systems,” in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, 2006, pp. 632–634.

[2] J. Cohen, “Constraint logic programming languages,” *Communications of the ACM*, vol. 33, no. 7, pp. 52–68, 1990.

[3] P. Van Hentenryck, “Constraint satisfaction in logic programming,” 1989.

[4] J. A. Nelder and R. Mead, “A simplex method for function minimization,” *The computer journal*, vol. 7, no. 4, p. 308, 1965.

[5] C. U. Smith and L. G. Williams, *Performance Solutions: a practical guide to creating responsive, scalable software*, vol. 34. Addison-Wesley Boston, MA;, 2002.

[6] D. C. Schmidt, “Guest editor’s introduction: Model-driven engineering,” *Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[7] S. Kent, “Model driven engineering,” in *Integrated Formal Methods*, 2002, pp. 286–298.

[8] A. Van Deursen, P. Klint, and J. Visser, “Domain-specific languages: An annotated bibliography,” *ACM Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[9] S. Kelly and J. P. Tolvanen, *Domain-specific modeling: enabling full code generation*. Wiley-IEEE Computer Society Pr, 2008.

[10] Á. Lédeczi et al., “Composing domain-specific design environments,” *Computer*, vol. 34, no. 11, pp. 44–51, 2001.

[11] S. Paunov, J. Hill, D. Schmidt, S. D. Baker, and J. M. Slaby, “Domain-specific modeling languages for configuring and evaluating enterprise DRE system quality of service,” in *Engineering of Computer Based Systems, 2006. ECBS 2006. 13th Annual IEEE International Symposium and Workshop on*, p. 10–pp.

[12] J. M. Slaby, S. Baker, J. Hill, and D. C. Schmidt, “Applying system execution modeling tools to evaluate enterprise distributed real-time and embedded system QoS,” 2006.

[13] J. White, A. Nechypurenko, E. Wuchner, and D. C. Schmidt, “Intelligence frameworks for assisting modelers in combinatorically challenging domains,” in *Proceedings of the Workshop on Generative Programming and Component Engineering for QoS Provisioning in Distributed Systems at the Fifth International Conference on Generative Programming and Component Engineering (GPCE 2006)*, 2006.

[14] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt, “A platform-independent component modeling language for distributed real-time and embedded systems,” in *Real Time and Embedded Technology and Applications Symposium, 2005. RTAS 2005. 11th IEEE*, 2005, pp. 190–199.

[15] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt, “A platform-independent component modeling language for distributed real-time and embedded systems,” *Journal of Computer and System Sciences*, vol. 73, no. 2, pp. 171–185, 2007.

[16] J. H. Hill, S. Tambe, and A. Gokhale, “Model-driven engineering for development-time QoS validation of component-based software systems,” in *Engineering of Computer-Based Systems, 2007. ECBS’07. 14th Annual IEEE International Conference and Workshops on the*, 2007, pp. 307–316.

[17] N. Wang et al., “Total quality of service provisioning in middleware and applications,” *Microprocessors and Microsystems*, vol. 27, no. 2, pp. 45–54, 2003.

[18] D. C. Schmidt, “Middleware for real-time and embedded systems,” *Communications of the ACM*, vol. 45, no. 6, pp. 43–48, 2002.
[19] J. M. Voas, “Certifying off-the-shelf software components,” *Computer*, vol. 31, no. 6, pp. 53–59, 1998.
[20] A. Gokhale, D. C. Schmidt, B. Natarajan, and N. Wang, “Applying model-integrated computing to component middleware and enterprise applications,” *Communications of the ACM*, vol. 45, no. 10, pp. 65–70, 2002.
[21] J. D. Poole, “Model-driven architecture: Vision, standards and emerging technologies,” in *Workshop on Metamodeling and Adaptive Object Models, ECOOP*, 2001.
[22] D. Sabin and E. C. Freuder, “Configuration as composite constraint satisfaction,” in *Proceedings of the Artificial Intelligence and Manufacturing Research Planning Workshop*, 1996, pp. 153–161.
[23] D. Benavides, P. Trinidad, and A. Ruiz-Cortés, “Automated reasoning on feature models,” in *Advanced Information Systems Engineering*, 2005, pp. 491–503.
[24] K. Czarnecki, S. Helsen, and U. Eisenecker, “Staged configuration through specialization and multilevel configuration of feature models,” *Software Process: Improvement and Practice*, vol. 10, no. 2, pp. 143–169, 2005.
[25] J. White, B. Dougherty, and D. C. Schmidt, “Selecting highly optimal architectural feature sets with Filtered Cartesian Flattening,” *Journal of Systems and Software*, vol. 82, no. 8, pp. 1268–1284, 2009.
[26] M. Akbar, E. Manning, G. Shoja, and S. Khan, “Heuristic solutions for the multiple-choice multi-dimension knapsack problem,” *Computational Science-ICCS 2001*, pp. 659–668, 2001.
[27] J. White, B. Doughtery, and D. C. Schmidt, “Ascent: An algorithmic technique for designing hardware and software in tandem,” *IEEE Transactions on Software Engineering*, pp. 838–851, 2010.
[28] J. White, D. C. Schmidt, and S. Mulligan, “The generic eclipse modeling system,” in *Model-Driven Development Tool Implementer’s Forum, TOOLS*, 2007, vol. 7.
[29] J. White and D. C. Schmidt, “Fireant: A tool for reducing enterprise product line architecture deployment, configuration, and testing costs,” in *Engineering of Computer Based Systems, 2006. ECBS 2006. 13th Annual IEEE International Symposium and Workshop on*, 2006, p. 2–pp.
[30] H. Beitollahi and G. Deconinck, “Fault-tolerant partitioning scheduling algorithms in real-time multiprocessor systems,” 2006.
[31] M. Mikic-Rakic and N. Medvidovic, “Architecture-level support for software component deployment in resource constrained environments,” *Component Deployment*, pp. 493–502, 2002.
[32] N. R. C. (US). S. C. for the D. S. of C. Aeronautics, N. R. C. (US). D. on Engineering, and P. Sciences, *Decadal survey of civil aeronautics: foundation for the future*. National Academies Press, 2006.
[33] D. Seto, J. P. Lehoczky, L. Sha, and K. G. Shin, “On task schedulability in real-time control systems,” in *rtss*, 1996, p. 13.
[34] S. Lauzac, R. Melhem, and D. Mossé, “Comparison of global and partitioning schemes for scheduling rate monotonic tasks on a multiprocessor,” in *Real-Time Systems, 1998. Proceedings. 10th Euromicro Workshop on*, 1998, pp. 188–195.

[35] A. A. Bertossi, L. V. Mancini, and F. Rossini, “Fault-tolerant rate-monotonic first-fit scheduling in hard-real-time systems,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 10, no. 9, pp. 934–945, 1999.
[36] A. Burchard, J. Liebeherr, Y. Oh, and S. H. Son, “New strategies for assigning real-time tasks to multiprocessor systems,” *Computers, IEEE Transactions on*, vol. 44, no. 12, pp. 1429–1442, 1995.
[37] S. K. Dhall and C. Liu, “On a real-time scheduling problem,” *Operations Research*, pp. 127–140, 1978.
[38] C. L. Liu and J. W. Layland, “Scheduling algorithms for multiprogramming in a hard-real-time environment,” *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.
[39] D. De Niz and R. Rajkumar, “Partitioning bin-packing algorithms for distributed real-time systems,” *International Journal of Embedded Systems*, vol. 2, no. 3, pp. 196–208, 2006.
[40] J. Kennedy and R. Eberhart, “Particle swarm optimization,” in *Neural Networks, 1995. Proceedings., IEEE International Conference on*, 1995, vol. 4, pp. 1942–1948.
[41] J. P. Lehoczky, L. Sha, J. K. Strosnider, and others, “Enhanced aperiodic responsiveness in hard real-time environments,” in *Proceedings of the IEEE Real-Time Systems Symposium*, 1987, pp. 261–270.
[42] L. Sha and J. B. Goodenough, “Real-time scheduling theory and Ada,” *Computer*, vol. 23, no. 4, pp. 53–62, 1990.
[43] J. K. Strosnider and T. E. Marchok, “Responsive, deterministic IEEE 802.5 token ring scheduling,” *Real-Time Systems*, vol. 1, no. 2, pp. 133–158, 1989.
[44] A. Carzaniga, A. Fuggetta, R. S. Hall, A. Van Der Hoek, D. Heimbigner, and A. L. Wolf, “A characterization framework for software deployment technologies,” *Dept. of Computer Science, University of Colorado, Tech. Rep., April*, 1998.
[45] J. A. Stankovic, “Strategic directions in real-time and embedded systems,” *ACM Computing Surveys (CSUR)*, vol. 28, no. 4, pp. 751–763, 1996.
[46] W. Damm, A. Votintseva, A. Metzner, B. Josko, T. Peikenkamp, and E. B\öde, “Boosting re-use of embedded automotive applications through rich components,” *Proceedings of Foundations of Interface Technologies*, vol. 2005, 2005.
[47] J. White, B. Dougherty, C. Thompson, and D. C. Schmidt, “ScatterD: Spatial Deployment Optimization with Hybrid Heuristic,” in *Evolutionary Algorithms*, 2010.
[48] C. M. Fonseca and P. J. Fleming, “Genetic Algorithms for Multiobjective Optimization: Formulation, Discussion and Generalization,” in *Genetic Algorithms*, 1993, p. 416.
[49] R. Poli, J. Kennedy, and T. Blackwell, “Particle swarm optimization,” *Swarm intelligence*, vol. 1, no. 1, pp. 33–57, 2007.
[50] M. Joseph and P. Pandya, “Finding response times in a real-time system,” *The Computer Journal*, vol. 29, no. 5, p. 390, 1986.
[51] D. Nurmi et al., “The eucalyptus open-source cloud-computing system,” in *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2009, pp. 124–131.
[52] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, J. Obbink, and K. Pohl, “Variability issues in software product lines,” *Software Product-Family Engineering*, pp. 303–338, 2002.
[53] K. C. Kang, “Feature-oriented domain analysis (FODA) feasibility study,” DTIC Document, 1990.
[54] V. Kumar, “Algorithms for constraint-satisfaction problems: A survey,” *AI magazine*, vol. 13, no. 1, p. 32, 1992.

[55] "Computer Center PowerNap Plan Could Save 75 Percent Of Data Center Energy." [Online]. Available: http://www.sciencedaily.com/releases/2009/03/090305164353.htm. [Accessed: 08-Sep-2011].

[56] D. Meisner, B. T. Gold, and T. F. Wenisch, "PowerNap: eliminating server idle power," *ACM SIGPLAN Notices*, vol. 44, no. 3, pp. 205–216, 2009.

[57] E. S. Rubin, A. B. Rao, and C. Chen, "Comparative assessments of fossil fuel power plants with CO2 capture and storage," 2004.

[58] R. Hankey, "Electric Power Monthly." [Online]. Available: http://www.eia.gov/cneaf/electricity/epm/epm_sum.html. [Accessed: 08-Sep-2011].

[59] A. Berl et al., "Energy-efficient cloud computing," *The Computer Journal*, vol. 53, no. 7, p. 1045, 2010.

[60] L. Liu et al., "GreenCloud: a new architecture for green data center," in *Proceedings of the 6th international conference industry session on Autonomic computing and communications industry session*, 2009, pp. 29–38.

[61] A. Bateman and M. Wood, "Cloud computing," *Bioinformatics*, vol. 25, no. 12, p. 1475, 2009.

[62] A. Beloglazov and R. Buyya, "Energy efficient allocation of virtual machines in cloud data centers," in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, 2010, pp. 577–578.

[63] R. Buyya, A. Beloglazov, and J. Abawajy, "Energy-Efficient management of data center resources for cloud computing: A vision, architectural elements, and open challenges," *Arxiv preprint arXiv:1006.0308*, 2010.

[64] S. Hazelhurst, "Scientific computing using virtual high-performance computing: a case study using the Amazon elastic computing cloud," in *Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology*, 2008, pp. 94–103.

[65] G. Singh, P. J. Young, N. C. Rowe, and T. S. Anderson, "Inexpensive seismic sensors for early warning of military sentries," in *Proceedings of SPIE*, 2010, vol. 7706, p. 77060J.

[66] C. E. Perkins and E. M. Royer, "Ad-hoc on-demand distance vector routing," in *Mobile Computing Systems and Applications, 1999. Proceedings. WMCSA'99. Second IEEE Workshop on*, 1999, pp. 90–100.

[67] C. E. Perkins and P. Bhagwat, "Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers," *ACM SIGCOMM Computer Communication Review*, vol. 24, no. 4, pp. 234–244, 1994.

[68] T. B\äck, *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford University Press, USA, 1996.

[69] R. Simmons et al., "Coordinated deployment of multiple, heterogeneous robots," in *Intelligent Robots and Systems, 2000.(IROS 2000). Proceedings. 2000 IEEE/RSJ International Conference on*, 2000, vol. 3, pp. 2254–2260.

[70] M. C. Bastarrica, A. A. Shvartsman, and S. A. Demurjian, "A Binary Integer Programming Model for Optimal Object Distribution," in *OPODIS*, 1998, pp. 211-226.

[71] J. N. Hooker, "Planning and scheduling by logic-based benders decomposition," *OPERATIONS RESEARCH-BALTIMORE THEN LINTHICUM-*, vol. 55, no. 3, p. 588, 2007.

[72] J. R. Koza, *On the programming of computers by means of natural selection*, vol. 1. MIT press, 1996.
[73] M. Armbrust et al., “A view of cloud computing,” *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.
[74] J. M. Maciejowski, *Predictive control: with constraints*. Pearson education, 2002.
[75] S. L. Chung, S. Lafortune, and F. Lin, “Limited lookahead policies in supervisory control of discrete event systems,” *Automatic Control, IEEE Transactions on*, vol. 37, no. 12, pp. 1921–1935, 1992.
[76] S. Abdelwahed, S. Neema, J. Loyall, and R. Shapiro, “A hybrid control design for QoS management,” in *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, 2003, pp. 366–369.
[77] S. Abdelwahed, N. Kandasamy, and S. Neema, “Online control for self-management in computing systems,” in *10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS’04)*, 2004, p. 368.
[78] N. Kandasamy, S. Abdelwahed, and M. Khandekar, “A hierarchical optimization framework for autonomic performance management of distributed computing systems,” 2006.
[79] D. Kusic, N. Kandasamy, and G. Jiang, “Approximation modeling for the online performance management of distributed computing systems,” *World*, vol. 1200, p. 1400, 1998.
[80] S. Abdelwahed, J. Bai, R. Su, and N. Kandasamy, “On the application of predictive control techniques for adaptive performance management of computing systems,” *Network and Service Management, IEEE Transactions on*, vol. 6, no. 4, pp. 212–225, 2009.
[81] S. Abdelwahed, N. Kandasamy, and S. Neema, “A control-based framework for self-managing distributed computing systems,” in *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, 2004, pp. 3–7.
[82] N. Roy, A. Dubey, A. Gokhale, and L. Dowdy, “A Capacity Planning Process for Performance Assurance of Component-based Distributed Systems,” in *Proceeding of the second joint WOSP/SIPEW international conference on Performance engineering*, 2011, pp. 259–270.
[83] N. Roy, A. Dubey, and A. Gokhale, “Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting,” in *IEEE 4th International Conference on Cloud Computing*, Washington, D.C, USA, 2011, pp. 500-507.
[84] C. D. Gill et al., “Integrated adaptive QoS management in middleware: A case study,” *Real-Time Systems*, vol. 29, no. 2, pp. 101–130, 2005.
[85] “OMG. Deployment and Configuration of Component-based Distributed Applications.” Object Management Group formal/2006-04-04, Apr 2006.
[86] G. Deng, J. Balasubramanian, W. Otte, D. Schmidt, and A. Gokhale, “DAnCE: a QoS-enabled component deployment and configuration engine,” *Component Deployment*, pp. 67–82, 2005.
[87] W. R. Otte, A. Gokhale, and D. C. Schmidt, “Predictable Deployment in Component-based Enterprise Distributed Real-time and Embedded Systems,” presented at the CBSE ’11, Boulder, CO, USA, 2011.
[88] A. Arulanthu, C. O’Ryan, D. Schmidt, M. Kircher, and J. Parsons, “The design and performance of a scalable ORB architecture for CORBA asynchronous messaging,” in *Middleware 2000*, 2000, pp. 208–230.

[89] J. White, B. Kolpackov, B. Natarajan, and D. C. Schmidt, “Reducing application code complexity with vocabulary-specific XML language bindings,” in *Proceedings of the 43rd annual Southeast regional conference-Volume 2*, 2005, pp. 281–287.

[90] W. Otte, D. Schmidt, and A. Gokhale, “Towards an Adaptive Deployment and Configuration Framework for Component-based Distributed Systems,” in *9th Workshop on Adaptive and Reflective Middleware (ARM ’10)*, Begaluru, India, 2010.

[91] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[92] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley, 2000.

[93] V. Subramonian et al., “The design and performance of component middleware for QoS-enabled deployment and configuration of DRE systems,” *Journal of Systems and Software*, vol. 80, no. 5, pp. 668–677, 2007.

Appendix A. List of Acronyms

ACE	Adaptive Communication Environment
AMI	Asynchronous Method Invocation
AMP	ASCENT Modeling Platform
ANT	Another Neat Tool
API	Application Programming Interface
ARMA	AutoRegressive Moving Average
ASCENT	Allocation baSed Configuration ExploratioN Technique
ATLAS	Automatically Tuned Linear Algebra Software
BLITZ	Bin packing LocatIon Technique for processor minimiZation
BON2	Builder Object Network
CBMG	Customer Behavior Modeling Graphs
CBML	Component Behavior Modeling Language
CCM	CORBA Component Model
CDP	Component Deployment Plan
CDR	Common Data Representation
CLP	Constrained Linear Programming
CONST	Constraints Optimization System
COTS	Commercial Off The Shelf
CoWorkEr	Component Workload Emulator
CSP	Constraint Satisfaction Problem
CUTS	CoWorkEr Utilization Test Suite
D&C	Deployment and Configuration
DAnCE	Deployment and Configuration Engine
DoD	Department of Defense
DOM	Document Object Model
DRE	Distributed real-time embedded
DSML	Domain-Specific Modeling Languages
EC2	Elastic Compute Cloud
EJB	Enterprise Java Beans
EMF	Eclipse Modeling Framework
FCF	Filtered Cartesian Flattening
FCS	Future Combat Systems
FFD	First Fit Decreasing
GEMS	Generic Eclipse Modeling System
GIG	Global Information Grid
GME	Generic Modeling Environment
GMF	Graphical Modeling Framework
GUTS	GEMS Utilization Test Suite
IDL	Interface Definition Language
KP	Knapsack Problem

LE-DAnCE	Locality-Enhanced Deployment and Configuration Engine
MCKP	Multiple-Choice Knapsack Problem
MDA	Model Driven Architecture
MDE	Model-Driven Engineering
MDKP	Multi-Dimensional Knapsack Problem
MMKP	Multiple-choice Multi-dimensional Knapsack Problems
MVA	Mean Value Analysis
NAOMI	New Associative Object Model of Integration
NOMAD	NetwOrk MinimizAtion Depolyment
OCL	Object Constraint Language
OMG	Object Management Group
ORB	Object Request Broker
OS	Operating System
PICML	Platform Independent Component Modeling Language
PIM	Platform-Independent Model
PLA	Product-Line Architectures
PSM	Platform-Specific Models
PSO	Particle Swarm Optimization
QA	Quality Assurance
QoS	Quality of Service
RTA	Response Time Analysis
SAX	Simple API for XML
ScatterD	Scatter Deployment Algorithm
SCORCH	Smart Cloud Optimization for Resource Configuration Handling
SEM	System Execution Modeling
SISPI	Software-Intensive Systems Productivity Initiative
SLA	Service Level Agreement
SOA	Service-Oriented Architecture
SPRUCE	Software PRoducibility Collaboration and Experimentation Environment
TAO	The ACE ORB
UAV	Unmanned Aerial Vehicle
UML	Unified Modeling Language
URL	Uniform Resource Locator
WML	Workload Modeling Language
XSC	XML Schema Compiler